Static Detection of Uninitialized Stack Variables in Binary Code

Behrad Garmany, Martin Stoffel, Robert Gawlik, and Thorsten Holz

Horst Görtz Institute for IT-Security (HGI)
Ruhr-Universität Bochum, Germany
{firstname.lastname}@rub.de

Abstract. More than two decades after the first stack smashing attacks, memory corruption vulnerabilities utilizing stack anomalies are still prevalent and play an important role in practice. Among such vulnerabilities, uninitialized variables play an exceptional role due to their unpleasant property of unpredictability: as compilers are tailored to operate fast, costly interprocedural analysis procedures are not used in practice to detect such vulnerabilities. As a result, complex relationships that expose uninitialized memory reads remain undiscovered in binary code. Recent vulnerability reports show the versatility on how uninitialized memory reads are utilized in practice, especially for memory disclosure and code execution. Research in recent years proposed detection and prevention techniques tailored to source code. To date, however, there has not been much attention for these types of software bugs within binary executables.

In this paper, we present a static analysis framework to find uninitialized variables in binary executables. We developed methods to lift the binaries into a knowledge representation which builds the base for specifically crafted algorithms to detect uninitialized reads. Our prototype implementation is capable of detecting uninitialized memory errors in complex binaries such as web browsers and OS kernels, and we detected 7 novel bugs.

1 Introduction

Memory corruption vulnerabilities are prevalent in programs developed in type-unsafe languages such as C and C++. These types of software faults are known since many years and discovering memory corruption bugs in binary executables has received a lot of attention for decades. Nevertheless, it is still an open research problem to efficiently detect vulnerabilities in binary code in an automated and scalable fashion. Especially *temporal* bugs seem to be a common problem in complex programs, an observation that the steady stream of reported vulnerabilities confirms [12–14, 43]. In practice, especially web browsers are often affected by temporal bugs and these programs suffer from use-after-free vulnerabilities, race conditions, uninitialized memory corruptions, and similar kinds of software vulnerabilities.

One specific challenge is the efficient detection of uninitialized memory errors, such as uninitialized stack variables. While such vulnerabilities got into the focus of several real-world attacks [4, 9–11, 14, 21], they still represent an attack vector that is not studied well and often overlooked in practice. The basic principle of such vulnerabilities is straightforward: if a variable is declared but not defined (i.e., not initialized properly) and used later on in a given program, then an attacker may abuse such a software fault as an attack primitive. The uninitialized variable may for example contain left-over information from prior variables in stale stack frames used during prior function calls. This information can be used to disclose memory and leak sensitive information, which can then be used by an attacker to bypass *Address Space Layout Randomization* (ASLR) or other defenses. In the worst case, an attacker can control the content of an uninitialized variable and use it to execute arbitrary code of her choice, hence fully compromising the program. Uninitialized memory errors represent a vulnerability class that often affects complex, real-world programs: for example, at the 2016 edition of the annual *pwn2own* contest, Microsoft's *Edge* web browser fell victim to an uninitialized stack variable [4]. As a result, this vulnerability was enough to exploit the memory corruption vulnerability and gain full control over the whole program. Similarly, an uninitialized structure on the stack was used in the *pwn2own* contest 2017 to perform a guest-to-host privilege escalation in VMware [21].

The detection of uninitialized variables in an automated way has been studied for software whose source code is available [17,22,24]. The urge for such systems, especially targeting the stack, is also addressed by recent research through tools like SafeInit [30] or UniSan [27]. These systems set their main focus on prevention and also rely on source code.

In practice, however, a lot of popular software is unfortunately proprietary and only available in binary format. Hence, if source code is unavailable, we need to resort to binary analysis. The analysis of binary code, on the other hand, is much more challenging since some of the context information gets lost during the compilation phase. The loss of data and context information (e.g, names, types, and structures of data are no longer available) hampers analysis and their reconstruction is difficult [20, 23, 39]. Thus, the development of precise analysis methods is more complicated without this information. Addressing this issue, we are compelled to consider every statement in the assembly code as it might relate to uninitialized memory of stack variables.

In this paper, we address this challenge and propose an automated analysis system to statically detect uninitialized stack variables in binary code. Since dynamic analysis methods typically lack comprehensive coverage of all possible paths, we introduce a novel static analysis approach which provides full coverage, at the cost of potentially unsound results (i.e., potential false positive and false negatives). However, unveiling potential spots of uninitialized reads and covering the whole binary poses a more attractive trade-off given the high value of detecting novel vulnerabilities. Note that the information obtained by our

approach can further serve in a dynamic approach, e.g., to automatically verify each warning generated by our method.

Our analysis is performed in two phases: First, we designed a framework to lift binary software into an intermediate representation, which is further transformed into a knowledge base that serves our *Datalog* programs. We opted for Datalog given that this declarative logic programming language enables us to efficiently query our deductive database that contains facts about the binary code. Based on Datalog, we then devised an accurate points-to analysis which is both flow- and field-sensitive. More specifically, with points-to information we have explicit information on indirect writes and reads that are connected to passed pointers. This allows us to track the indirect read or write back to the specific calling context where the points-to information is further propagated and incorporated in our knowledge base. This analysis step builds up the conceptual structure on which our dataflow algorithms to detect uninitialized variables operate.

To demonstrate the practical feasibility of the proposed method, we implemented a prototype which is tailored to detect uninitialized stack variables. Our results show that we can successfully find all uninitialized stack vulnerabilities in the Cyber Grand Challenge (CGC) binaries. In addition, we detected several real-world vulnerabilities in complex software such as web browsers and OS kernel binaries. Finally, our prototype is able to detect and pinpoint new and previously unknown bugs in programs such as *objdump*, and *gprof*.

In summary, our main contributions in this paper are:

- We design and implement an automated static analysis approach and introduce several processing steps which enable us to encode the complex data flow within a given binary executable to unveil unsafe zones in the control flow graph (i.e., basic blocks in which potential uninitialized reads might occur).
- We present a flow-, context- and field-sensitive analysis approach built on top of these processing steps, suitable for large-scale analysis of binary executables to detect uninitialized reads in a given binary executable.
- We evaluate and demonstrate that our analysis framework can detect both vulnerabilities in synthetic binary executables and complex, real-world programs. Our results show that the framework is capable of finding new bugs.

2 Uninitialized Stack Variables

Stack variables are local variables stored in the stack frame of a given function. A function usually allocates a new stack frame during its prologue by decreasing the stack pointer and setting up a new frame pointer that points to the beginning of the frame. Depending on the calling convention, either the caller or callee take care of freeing the stack frame by increasing the stack pointer and restoring the old frame pointer. For example, in the `stdcall` calling convention, the callee is responsible for cleaning up the stack during the epilogue. It is important to note that data from deallocated stack frames are *not* automatically overwritten

during a function's prologue or epilogue. This, in particular, means that old (and thus stale) data can still be present in a newly allocated stack frame. A stack variable that is not initialized properly hence contains old data from earlier, deallocated stack frames. Such a variable is also called *uninitialized*. An uninitialized stack variable can lead to undefined behavior, not at least due to its unpleasant property that the program does not necessarily crash upon such inputs. In practice, uninitialized variables can be exploited in various ways and pose a serious problem [4, 9–11, 14, 28]. They usually contain junk data, but if an attacker can control these memory cells with data of her choice, the software vulnerability might enable arbitrary code execution.

To tackle this problem, the compiler can report uninitialized stack variables at compile time for intraprocedural cases. Unfortunately, interprocedural cases are usually not taken into account by compilers. This lies in the nature of compilers which need to be fast and cannot afford costly analysis procedures. Even for optimization purposes, past research reveals that the benefits of extensive interprocedural analyses are not large enough to be taken account of in compilers [35].

3 Design

In the following, we provide a comprehensive overview of our static analysis framework to detect uninitialized stack variables in binary executables. Our analysis is divided into two processing stages. In a pre-processing step, we lift the binary into an IL and transform each function into SSA with respect to registers. The transformed functions are translated into Datalog facts which serve as our extensional database or *knowledge base*.

Based on this database, we then perform an interprocedural points-to analysis with respect to stack pointers. The analysis also results in information about pointers that are passed as arguments to functions and hence we can determine those pointers that enter a new function context. The reconstructed information about indirect definitions and uses of stack locations is used in a post-processing state in which we determine *safe zones* for each stack access. Safe zones consist of *safe basic blocks*, i.e., a use in these blocks with respect to the specific variable is covered by a definition on all paths. Stack accesses outside their corresponding safe zone produce warnings. For each variable, we determine a safe zone in its corresponding function context.

Our dataflow algorithms propagate information about safe zones from callers to callees and vice versa. If a path exists from the entry of a function to the use of a variable, which avoids the basic blocks of its safe zone, then the stack access is flagged *unsafe*. If a potentially uninitialized parameter is passed to a function, we check if the exit node of the function belongs to its safe zone. This in particular means that each path from the entry point of the function to the leaf node is covered by a definition (i.e., initialization) of the variable. We propagate this information back to the call sites, i.e., the fallthrough basic block at the call site is added to the safe zone of the variable in the context of

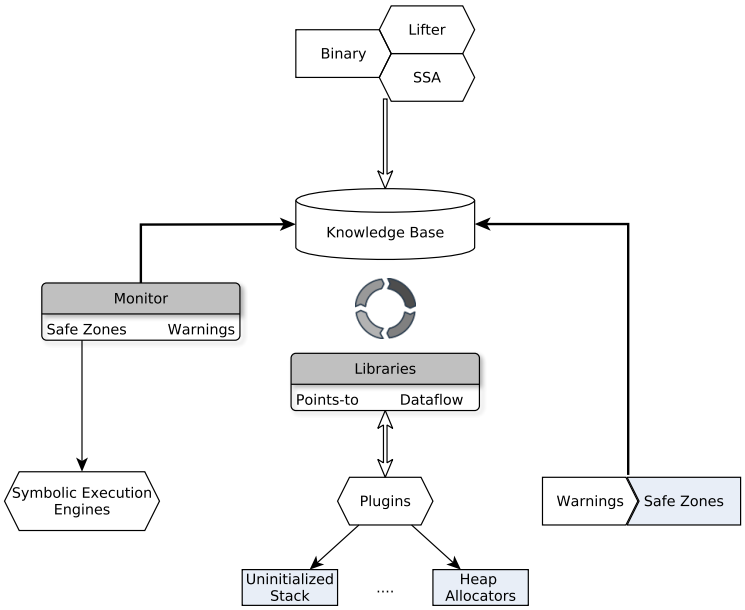

Fig. 1: Architecture implemented by our prototype.

the caller. This information in turn is further propagated and used at other call sites. Figure 1 shows the architecture implemented by our prototype. Our design allows to attach different checker plugins into the system that work and enrich the same knowledge base with valuable information. Each plugin can be run in parallel. Whenever a new information enters the knowledge base, the plugins adapt to it.

Warnings and safe zones are outputs of the analysis phase and put into the knowledge base. A monitor observes changes made to safe zones and warnings to either spawn the Datalog algorithms or a symbolic execution engine. The symbolic execution engine tackles the path sensitivity and is fed with safe zones of each stack variable. The aim of the symbolic execution engine is to reach the warning, i.e., a potential uninitialized read, by avoiding the safe zone of that variable. The whole procedures cycle, i.e., each component contributes to the knowledge base which in turn is consumed by other components to adapt.

Our current prototype is tailored towards uninitialized stack variables. However, as the plugin system suggests, we are able to enrich the analyses with heap information (see § 5.3). In the next sections, we explain the individual analysis steps in detail and present examples to illustrate each step.

3.1 Stack Pointer Delta

On Intel x86 and many other instruction set architectures, the stack pointer is used to keep a reference to the top of the stack. The stack is usually accessed

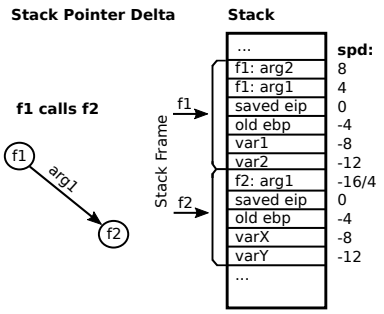

Fig. 2: Stack Pointer Delta (spd) addressing.

in relation to the current stack pointer. On Intel x86, the frame pointer is additionally linked to the stack pointer and keeps a reference to the beginning of the current stack frame. A function stack frame can, therefore, be located on different stack addresses depending on its stack pointer value. Because of this, we can not refer to a stack address directly. Instead, we use a delta value depending on the stack pointer. We refer to this addressing type as *stack pointer delta addressing*. A stack frame always starts at delta zero and grows towards negative values. Therefore, local variables are located at negative offsets, while passed arguments through the stack reside at positive offsets. We handle the *fastcall* calling convention for both x64 and x86 architectures in a generic way by means of Datalog rules . To simplify the analysis, we rebase all memory accesses on the stack to be relative to the stack pointer instead of using the frame pointer.

Definition 1. *Let S be the set of all stack variables. Each $s \in S$ is a tuple of the form (spd_s, fld_s) where spd_s is the stack pointer delta of s and fld an optional field value added to the base address of the underlying memory object.*

Figure 2 illustrates this concept. Function `f1` calls `f2`. The resulting stack is shown on the right-hand side. In each stack frame the saved EIP value is mapped to the delta value zero. A direct access to var1 inside function `f1` is associated with $(-8, 0)$, since it is at delta -8. Variable *varX* is also at delta value -8 but inside function `f2`. Each delta value is associated with the corresponding function in which the access occurs.

3.2 Points-To Analysis for Binaries

Analyzing binary machine code poses many challenges to overcome. The lack of type information forces us to analyze each IL statement. To overcome the complicated arrangement of work list algorithms, we decided to opt for a declarative approach. Rather than solving the problem by an imperative algorithm, we describe the problem and let the solver perform the evaluation strategy. Therefore,

Input: EDB
Output: Points-To Facts

① VPtsTo(V1, SPD, Addr, Ctx) ←
StackPointer(V1, Addr, SPD).

② VPtsTo(V1, SPD, Addr, Ctx) ←
Assign(V1,V2, Addr),
VptsTo(V2, SPD, _, Ctx).

③ VPtsTo(V1, SPD2, Addr, Ctx) ←
Load(V1, V2, Disp, Addr, Ctx),
VPtsTo(V2, SPD, Addr2, Ctx),
CanReach(Addr, Addr2, Ctx),
PointerPtsTo(SPD, Disp, SPD2).

④ PointerPtsTo(SPD,Disp,SPD, Ctx) ←
Store(V1, Disp, V2, Addr, Ctx),
VPtsTo(V1, SPD, _, Ctx),
VptsTo(V2, SPD, _, Ctx).

⑤ VPtsTo(Res, SPD+Value, Addr, Ctx) ←
BinOp(Op, Res, V1, V2, Addr, Ctx),
VPtsTo(V1, SPD, _, Ctx),
Constant(V2, Value, Addr).

⑥ VPtsTo(V2, SPD2, CalleeAddr, Callee) ←
Param(V1, Arg, Addr, Caller,
V2, CalleeAddr, Callee),
TranslateSPD(Arg, Callee, SPD2),
VptsTo(V1, _, _, Caller).

⑦ VPtsTo(V1, SPD, Addr, Ctx) ←
Phi(V1, PhiReg, Addr, Ctx),
VptsTo(V1, SPD, _, Ctx).

⑧ IndirectDef(V1, SPD, Addr) ←
Store(V1, Disp, V2, Addr, Ctx),
VptsTo(V1, SPD, _, Ctx).

⑨ IndirectUse(V1, SPD, Addr) ←
Load(V1, V2, Disp, Addr, Ctx),
VptsTo(V1, SPD, _, Ctx).

Algorithm 1: Points-To Analysis.

Variables:

$Addr$	Instruction address
V_i	Register or memory expression
SPD	Stack pointer delta value (spd) of the corresponding stack location
Ctx	Function name/context
Arg	Part of Param facts; describes which parameter we are dealing with ($1st, 2nd, \ldots$)

EDB Facts:

StackPointer	A fact that unifies V1 with the register that holds a pointer. The Addr of the instruction and the spd value of the stack location are stored in the corresponding variables, respectively.
Assign	Corresponds to a `mov` instruction in assembly
Load	Dereference from $V_2 + Disp$ and store it into V_1
Store	Store content of V_2 into memory at $V_1 + Disp$
Param	Describes a parameter pass at the call site ($Addr$) from actual V_1 in the caller context to a formal V_2 in the callee context
TranslateSPD	Translates the spd value of the parameters in the context of the callee and vice versa. In x86 first parameter has spd value 4, second has 8 etc.
BinOp	Describes a binary operation where Op is applied on V_1 and V_2 and the result is stored in Res.
Constant	Describes a constant value used in a binary operation
Phi	Corresponds to an SSA phi assignment. $PhiReg$ is bound to registers in the phi expression

Fig. 3: Variables and EDB facts.

we utilize an Andersen-style algorithm [1] in Datalog which is flow- and field-sensitive. Our algorithm is inspired by recent research done by Smaragdakis et al. [40,41]. They show how context, flow, and field sensitivity can be achieved in large-scale through a Datalog-based approach. We adapted their approach and tailored our algorithms for binary analysis.

Each information about memory loads, stores, assignments, arithmetic operations, control flow, and parameter passes which is expressed in terms of the IL, is extracted into an extensional database (EDB). For each binary, an EDB is produced which represents a knowledge base of a priori facts.

A simplified version of our approach delivers the idea and is presented by Algorithm 1. The Datalog algorithm is fed with the EDB which builds the base for Datalog rules to derive new facts. These new facts build the intensional database (*IDB*). The *IDB* and *EBD* form our *knowledge base*. In Figure 3, we summarize the facts and variables used in Algorithm 1. Some rules are left-out for the sake of brevity.

Datalog. To better understand what Algorithm 1 does, we refer the reader to common literature on logic programming. Datalog is a declarative logic programming language that has its roots in the database field [5] and its general purpose is to serve as a query language for large, complex databases. Datalog programs run in polynomial time and are guaranteed to terminate. Conventional Datalog uses a Prolog-like notation, but with simpler semantics and without the data structures that Prolog provides. Its approach to resolve new facts is close to what dataflow algorithms do with arrangements of worklist algorithms. It strives a set-oriented approach which we require, rather than a goal-oriented approach as done in Prolog. A Datalog program consists of facts and rules which are represented as *Horn clauses* of the form: $P_0 :- \quad P_1, \ldots, P_n$, where P_i is a literal of the form $p(x_1, \ldots, x_k)$ such that p is a predicate and the x_j are terms. The left-hand side of the clause is called *head*; the right-hand side is called *body*. A clause is true when each of its literals in the body are true.

Rules. Refer to Algorithm 1. The predicate `VPtsTo` stands for the points-to set of a variable. Rule ② specifies the following: Given an assignment from `V2` to `V1` at a specific address, include the points-to set of `V2` into that of `V1`. Rule ⑤ specifies a case of derived pointers: Given a binary operation such that `Res = V1 + V2`, where `V1` is a pointer, check if `V2` is a constant. If the conditions hold, then `Res` points to a stack location with a stack pointer delta of `SPD+Value`. Variable `Value` is grounded by the third fact in the body of this rule. With rule ⑤ we deduce a new points-to set that corresponds to a new stack location, which in turn is again used to derive new points-to set information by the recursive chain of rules. This procedure is performed on all rules until the process saturates, i.e., a fixpoint is reached where we do not gain additional facts. Another rule that deserves attention is rule ⑥. Here points-to information is tracked into the context of the callee. The rule specifies that if a stack pointer is passed as a parameter, then a new points-to set is created for that parameter in the context of the callee. If, for example, a stack pointer is passed as an argument, then `TranslateSPD` in the body of the rule gives us its stack pointer delta value in the context of the callee. A new points-to set is created for this argument which is basically a new derived fact. This fact, in turn, falls into the recursive chain to serve for deducing new facts. The procedure provides an on-demand approach to track arguments of a function, only when they are passed by reference. We achieve context-sensitivity by introducing tags at each call sites. These tags are linked with the parameters.

To illustrate the approach, let's assume that our analyzer runs over the following piece of code:

```
foo:
  0x8049000 mov eax, [esp+4]
  0x8049004 mov dword [eax], 0xff
  0x804900a mov [eax+4], eax
```

```
main:
 ...
 0x80490f0: lea ebx, [esp-0x30]
 0x80490f4: push ebx
 0x80490f5: call foo
```

At `0x80490f0-0x80490f4` a stack pointer is pushed onto the stack with a delta of `-0x30` which resides in `[esp+4]` in the context of `foo`. As the result of the preprocessing step, we have `Param` and `TranslateSPD` facts extracted into our EDB (see Figure 3). For this example, we have the facts `Param([esp_17], 1, 0x80490f5, "main", [esp+4], "foo")`, and `TranslateSPD(1, "foo", 4)`, where `[esp_17]` corresponds to the location the stack pointer is pushed to at `0x80490f4`. Since we are dealing with a passed stack pointer, our analysis derives `VptsTo([esp_17], -0x30, 0x80490f4. "main")`. By using rule ⑥, the points-to analysis can now deduce the fact `VPtsTo([esp+4], 4, 0x8049000, "foo")`. Note that the pointer is now considered to have a delta of 4 in the *new context*. We do this to keep track of pointers that are parameters, otherwise we lose focus on where the pointer might *originate* from.

With rule ②, we get the connection to `eax`, and with rule ③, ④ the connection of `[eax+4]` to the underlying memory object, i.e., a pointer to itself. By Definition 1, we refer to `[eax+4]` as $(4, 4)$ since the base points to a location with delta 4 and we are accessing the location that is 4 bytes apart from the base.

3.3 Safe Zones

In this section we describe our approach to determine if a given stack read is safe. With *safe* we refer to the property that a read is covered by its definitions on all paths. Each basic block where a *safe* read occurs is considered a *safe basic block*. Since we are dealing with different memory objects, the set of safe basic blocks is different for each object/variable. More formally, we define it as follows.

Definition 2. *Let $CFG = (V, E)$ be the control flow graph, S the set of all stack variables, and let $Defs = \{(spd, fld, bb_s) \mid bb_s \in V, (spd, fld) \in S\}$ be the set of all stack accesses that define the stack location (spd, fld) at bb_s. bb_s is called a safe basic block. Each edge that originates from bb_s is called a safe edge with respect to (spd, fld). Each safe edge is a tuple of the form (spd, fld, bb_s, bb_t) with $(bb_s, bb_t) \in E$. The set of all safe basic blocks with respect to (spd, fld) is called the safe zone of (spd, fld).*

Apparently, if all incoming edges to a basic block are *safe edges* with respect to some variable `(spd,fld)` then that basic block is a *safe basic blocks* for that variable. To determine *safe zones* of each variable we proceed as sketched in Algorithm 2. An unsafe read occurs if a path exists that avoids the safe zone, i.e., a path from the entry of the function to the *use* location which does not go through safe edges. Lines $17 - 21$ generalize this procedure for all stack accesses. If such a path does not exist, we flag the basic block as safe.

In its essence the algorithm does a reaching definition analysis for each stack variable and labels the basic blocks and edges accordingly. The initial process for

building safe zones is achieved through lines $7 - 13$. From each definition node with respect to the specific stack variable, the information about its definition is propagated further along its path in the control flow graph.

Each stack variable is associated with its own safe zone. Note that we do not use memory SSA as it introduces conflicts and complicates the points-to analysis. Hence, benefits of SSA in this manner are marginal.

1: Input: CFG $= (V, E)$, DEFS, USES
2: DOM (*dominator sets*)
3: Outputs: SAFEZONE, E (SAFEEDGES)
4: Let $E' =$ SAFEEDGES $= \{\}$
5: Let SAFEZONE $=$ DEFS
6: Let VARS $= \bigcup_{(spd,fld,bb) \in \text{DEFS} \cup \text{USES}} (spd, fld)$
7: **for each** $(spd, fld, bb) \in$ SAFEZONE **do**
8: $E' = E' \cup \{(spd, fld, bb, bb_x) \mid (bb, bb_x) \in E\}$
9: **for each** $bb_d \in \text{DOM}(bb)$ **do**
10: $E' = E' \cup \{(spd, fld, bb_d, bb_x) \mid (bb_d, bb_x) \in E\}$
11: SAFEZONE $=$ SAFEZONE $\cup \{(spd, fld, bb_d)\}$
12: **end for**
13: **end for**
14: Let UNSAFE $= \{\}$
15: **for each** $(spd, fld, bb) \in$ VARS **do**
16: **if** $\exists p =< bb_{start}, \ldots, bb_i, bb_j, \ldots, bb >$ with
17: $(spd, fld, bb_i, bb_j) \notin E', \forall bb_i, bb_j \in p, i \neq j$
18: **then** UNSAFE $=$ UNSAFE $\cup \{(spd, fld, bb)\}$
19: **else** SAFEZONE $=$ SAFEZONE $\cup \{(spd, fld, bb)\}$
20: **end for**

Algorithm 2: Sketch: Computation of Safe Zones

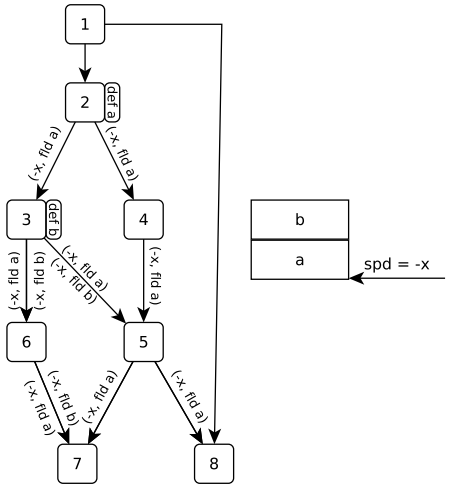

Fig. 4: Graphical representation of labeling safe edges.

Example 31. *Figure 4 illustrates the labeling of safe edges. Basic blocks* 2 *and* 3 *define variables* a *and* b*, respectively. Each use of the variable in these basic blocks is considered* safe*. Accordingly, each use of variable* b *in* $\{3, 6\}$ *is considered safe. At a stack pointer delta of* $-x$ *an access to variable* a *is possible. Its field/offset* $(fld\ a)$ *is zero. For an access to variable* b*,* fld b *is added. Safe Zone with respect to* $(-x, fld\ a)$ *consists of basic basic blocks* $\{2, 3, 4, 6, 5, 7\}$*. For* $(-x, fld\ b)$ *we have* $\{3, 6\}$*. Each use of the variable in these basic blocks is considered* safe*.*

3.4 Interprocedural Flow Analysis

During the data flow analysis, we propagate state information that concerns the initialization of passed arguments between caller and callee. Information about pointers is passed back and forth by the points-to analysis. We use this information to determine indirect accesses (see rule ⑧,⑨ in Algorithm 1). If a leaf node in the callee context is flagged *safe* with respect to a stack access, we flag the corresponding call sites as safe. This procedure propagates information back to the caller, extending the safe zone in the caller context. In turn, Algorithm 2

needs another run by using the new information and distinguish between unsafe and safe accesses to the stack. Previously unsafe accesses might turn to safe accesses through this process. This procedure is repeated until it saturates, i.e., no changes to the set of safe basic blocks.

Summaries: A common technique used in interprocedural static analysis is the use of summaries [37]. These summaries can be block summaries which gather the effects of a basic block, or function summaries that gather the effects of the whole function with respect to the variables of interest. Whenever a function call is encountered, these summaries are utilized and applied. The facts in Datalogs EDB and its deduced facts through rules in the IDB can be seen as such summaries. Whenever a function call is encountered, the analysis uses facts about the function that concern the variables of interest.

Multiple analyses plugins: As shown in Figure 1, our design has a plugin mechanism. All plugins operate on the same knowledge base. Plugins deduce and incorporate knowledge into the knowledge base which can transparently be tapped by other plugins and library routines. This, for instance, allows the *Uninitialized Stack* plugin to operate on information deduced by the *Heap Allocators* plugin. Each plugin can be run in parallel and whenever new information enter the knowledge base, the plugins adapt to it. Each change to the knowledge base with respect to warnings and safe zones is monitored.

Detecting Uninitialized Accesses: When the analysis reaches its fixpoint, all information about safe and unsafe zones with respect to all stack accesses is present. A stack access outside its safe zone causes a warning. Additionally, we track each use to its origin, i.e., in the case of a stack pointer, we track it to the call site where the pointer originates from.

3.5 Symbolic Execution

For each warning, we need to check if a satisfiable path exists to the use of the variable by avoiding its safe zone. We therefore need a mechanism for path-sensitivity in our analysis process. To tackle path sensitivity, we utilize under-constrained symbolic execution [32]. Under-constrained symbolic execution immensely improves the scalability by checking each function directly, rather than the whole program. Due to the lack of context before the call of the function under analysis some register/memory values are unconstrained, hence the under-constrained term.

For each variable that caused a warning, we feed the symbolic execution engine with information about its safe zones. Satisfiability is checked from the function entry to the flagged variable by avoiding the basic blocks in its safe zone. We start at each origin, i.e., the function where the stack variable originates from. To improve the scalability of the symbolic execution, we initially skip each function call. If a path is satisfiable then we might have an overapproximation, since some skipped function might have made a constraint become unsatisfiable.

For unsatisfiable paths, we look at the *unsat-core*, i.e. those constraints which have caused the path become unsatisfiable. A function that alters one of these variables in those constraints is then set free to be processed by the engine;

again in a similar fashion by first skipping calls in the new function context until we eventually reach a satisfiable state. The only difference is that we now force the engine to run into basic blocks that modify the variables that made our constraints become unsatisfiable. As a result, we basically overapproximate the set of satisfiable paths. Filtered warnings are removed as such in the knowledge base.

4 Implementation

Our prototype is implemented on top of the Amoco framework [44]. The decision for Amoco is favored due to its flexible IL. It allows us to extend its set of expressions. Each new expression transparently integrates and interplays with the standard set of expressions.

We retrieve the control-flow graph from the disassembler IDA Pro which is shown to be the most accurate [2]. Each basic block is transformed into an Amoco basic block. We extended Amoco to support SSA and implemented the algorithm proposed by Cytron et al. [15]. In particular, we adapted the concept of collectors that are described in Van Emmerik's work on decompilers [45]. Collectors can be seen as an instrumentation of the SSA algorithm. The algorithm proposed by Cytron et al. uses a stack of live definitions whenever a basic block is processed. This information is valuable to put into a knowledge base. For instance, we can instrument the algorithm to write a set of live definitions at call sites into our knowledge base which we use to translate SSA subscripted expressions back and forth between caller and callees. Due to SSA with respect to registers, we obtain partial flow sensitivity.

We built the symbolic execution on top of angr [38], a platform agnostic binary analysis framework. As Figure 1 indicates, we plan to attach more engines to our framework. This is motivated by the fact that each engine comes with advantages and its shortcomings, which we hope to compensate by combining different engines. The points-to analysis results are saved into a separate database. If extensions are needed, we can reuse this database and let Datalog evaluate new facts based on the new extensions.

5 Evaluation

In this section, we evaluate the prototype implementation of our analysis framework and discuss empircal results. Note that the analysis framework is by design OS independent. Our analyses were performed on a machine running with Intel Xeon CPUs E5-2667@2.90GHZ and 128GB RAM. The programs presented in Table 1 and Table 2 are compiled for *x86-64*. Our prototype is not limited to 64 bit, but also supports 32 bit binaries.

5.1 CGC Test Corpus

As a first step to obtain a measurement on how our approach copes with realistic, real-world scenarios, we evaluated our prototype over a set of Cyber Grand

Table 1: Analysis results for the relevant CGC binaries that contain an uninitialized memory vulnerability.

Binary	Functions	Facts	Pointer Facts	Stack Accesses	Unique Warnings
Hackman	70	46k	545	943	9
Accel	185	109k	2179	2057	33
TFTTP	58	30k	175	600	3
MCS	122	156k	860	2498	11
NOPE	105	57k	568	1378	8
Textsearch	90	50k	290	597	2
SSO	64	26k	204	650	4
BitBlaster	10	4k	42	95	1

Challenge (CGC) binaries which, in particular, contain a known uninitialized stack vulnerability. These CGC binaries are built to imitate real-world exploit scenarios and deliver enough complexity to stress out automated analysis frameworks. Patches of the vulnerabilities ease the effort to find the states of true positives and hence these binaries can serve as a ground truth for our evaluation. We picked those binaries from the whole CGC corpus that are documented to contain an uninitialized use of a stack variable as an exploit primitive and we evaluate our prototype with these eight binaries.

Table 1 shows our results for this CGC test setup. The third column of the table depicts the number of facts extracted from the binary building up the EDB. The fourth column shows the number of deduced pointer facts. The fifth column depicts the total number of stack accesses. The sixth column denotes the number of potential uninitialized stack variables grouped by their stack pointer delta value and their origin. This approach is similar to *fuzzy stack hashing* as proposed by Molnar et al. [31] to group together instances of the same bug.

For each of the eight binaries, we successfully detected the vulnerability. Each detected use of an uninitialized stack variable is registered, among which some might stem from the same origin. Therefore, we group those warnings by the stack pointer delta values of those stack variables from which they originate. The individual columns of Table 1 depict this process in numbers. We double-checked our results with the patched binaries to validate that our analysis process does not produce erroneous warnings for patched cases. For each patched binary, our analysis does not generate a warning for the vulnerabilities anymore.

5.2 Real-World Binaries

Beyond the synthetic CGC test cases, we also applied our analysis framework on real-world binaries. Table 2 summarizes our results for *binutils gnuplot*, and *ImageMagick*. The values in parentheses are manually verified bugs. Note that the number of warnings pinpointing the root cause and the potential flaw through an uninitialized variable is comparatively *small* to the number of all accesses. Additionally, our symbolic execution filter was able to reduce the warning rate by a factor of eight in our experiments. Despite the false positive rates which we discuss in the next sections, we strongly believe that the output of our prototype

Table 2: Analysis results for binutils-2.30, ImageMagick-6.0.8, gnuplot 5.2 patch-level 4. Number in parentheses denotes the number of verified bugs.

Binary	Functions	Facts	Pointer Facts	Stack Accesses	Unique Warnings
objdump	2.5k	>4M	>19k	23k	42 (2)
ar	2.4k	>3.2M	>16k	19k	24
as-new	2.2k	>2.9M	>9k	15k	29
gprof	2.3k	>3.8M	>16k	20k	34 (1)
cxxfilt	2.2k	>3.1M	>15k	18k	22
ld-new	2.8k	>3.8M	>16k	22k	15
strings	2.2k	>3.5M	>15k	19k	11
size	1.9k	>3.1M	>15k	19k	20
readelf	115	>107k	>5k	543	4
gnuplot	3k	>7.2M	>12	23k	54 (2)
Image Magick	6.5k	>24M	>31k	150k	168 (2)

is a valuable asset for an analyst and the time spent to investigate is worth the effort. The numbers are given in column 6 of Table 2. Overall, we found two recently reported vulnerabilities in *ImageMagick*, two previously unknown bugs in *objdump*, one unknown bug in *gprof*, and two bugs in *gnuplot*. A manual analysis revealed that the bugs in objdump and gprof are not security critical. The two bugs in gnuplot were fixed with the latest patchlevel at the time of writing.

Our analysis can also cope with complex programs such as web browsers, interpreters and even OS kernels in a platform-agnostic manner. We tested our framework on *MSHTML (Internet Explorer 8, CVE-2011-1346)*, *Samba (CVE-2015-0240)*, *ntoskrnl.exe (Windows kernel, CVE-2016-0040)*, *PHP (CVE-2016-7480)*, and *Chakra (Microsoft Edge, CVE-2016-0191)*. In each case, we successfully detected the vulnerability in accordance to the CVE case.

Error Handling Code: Our study on hundreds of warnings shows that many warnings are internally handled by the programs itself through error handling code. To address this problem, we implemented a plugin that checks—starting from the corresponding call site—if a return value might go into a sanitization process. We track the dataflow into a jump condition and measure the distance to the leaf node. If it is smaller than a threshold value of 3, we assume that the return value is sanitized and handled. This simple procedure works surprisingly well in most cases to shift the focus away from paths that run into error handling code.

5.3 Heap Allocations

User space programs use a variation of `malloc` for allocating memory dynamically. Performance-critical applications like browsers even come with their own custom memory allocators. To enable tracking of dynamic memory objects, we use a list of known memory allocators and enriched the knowledge base with pointer information. The points-to analysis grabs this information and deduces new facts. As a consequence we can observe a coherence between stack locations

and heap. While this is an experimental feature of our framework, it has proven itself valuable by pinpointing three uninitialized bugs in `gprof` and `objdump` which originate from the heap.

6 Discussion

The discovery of vulnerabilities for both the CGC binaries and real-world binaries demonstrates that our approach can successfully detect and pinpoint various kinds of uninitialized memory vulnerabilities. Our analysis is tailored to stack variables, with a design that is well-aligned with the intended purpose of a bug-detecting static analysis. However, it also comes with some limitations that are not currently tackled by our prototype and we discuss potential drawbacks of our approach in the following.

Heap: It is well known that analyzing the data flow on the heap is harder than data flow on the stack and in registers. To address this problem, for example Rinetzky and Sagiv proposed an approach to infer shape predicates for heap objects [36]. More recently, the topic of separation logic has garnered more attention as a general purpose shape analysis tool [34]. This is—among other reasons—due to the fact that aliasing analysis becomes much more difficult for real-world code that makes use of the heap as compared to the data flow that arises from stack usage. We account for all stack accesses under the reconstructed CFG. Hence, a stack variable which is initialized through a heap variable is supported by our approach. A points-to analysis needs to account for this interplay. Therefore, we implemented a component that adapts to the given set of points-to facts and tracks heap pointers originating from known heap allocators (see § 5.3). This procedure is by design not sound, however the discovered bugs which originated from the heap were found by using this approach.

Many performance-critical applications like browsers have their own custom memory allocators which poses a problem to address. However, there is work on this field with promising results as shown in recent research done by Chen et al. [6, 8].

False Positives/False Negatives: Many analyzers come with a set of strategies to deal with the number of warnings by, for instance, checking the feasibility of paths. Each strategy is usually tied to certain aspects of the problem, an approach which we adapted and discussed in the last sections to tackle false positives. However, we are dealing with many *undecidable* problems here, i.e., the perfect disassembly and the fact that detection of uninitialized variables itself is *undecidable* in general.

Aggressive compiler optimizations can also pose a burden on the disassembly process as they can facilitate the problem of unpredictable control flow. Even for a state-of-the-art tool like IDA Pro, control-flow analysis is hampered by unpredictability and indirect jumps. Points-to information can resolve some of these indirect jumps [16]. However, its demand for context sensitivity is expensive for

large applications. Programs that contain recursion further restrict the capabilities of static analysis, as shown by Reps [33]. A combination with dynamic approaches might prove itself valuable and information derived by them can be incorporated into our knowledge base. Recall, that each change in the knowledge base is adapted transparently, i.e., facts are removed, added and deduced constantly by adding or removing information from it.

A valuable feature for any analyzer is the question of *soundness*. A fully sound analysis is hard to achieve in practice due to code that we cannot analyze (e.g., libraries which are not modeled or dynamically dispatched code where we might loose sight). A sound analysis needs a perfect disassembly resulting in a perfect CFG, which is an *undecidable* problem. Therefore, false negatives are not avoidable. Similar source code based systems use the term *soundiness* [26] as they can guarantee soundness for specific parts of the code under analysis only [29].

We strongly believe that if an analyzer finds enough errors to repay the cost of studying its output, then the analyzer will be a valuable and cost-effective instrument in practice.

7 Related Work

The development of methods for static analysis spans a wide variety of techniques. Modern compilers like the GNU-Compiler, MSVC, or Clang can report uninitialized variables during compile time. They utilize the underlying compiler framework to implement an intraprocedural analysis to detect potential uninitialized variables. As discussed earlier, compilers are trimmed to run fast, and the analysis time for costly interprocedural algorithms is not desired. For optimization purposes, the benefits of extensive interprocedural analyses might not be desirable to apply [35].

Flake was one of the first to discuss attacks against overlapping data [18], an attack vector closely related to our work on uninitialized memory reads. His presentation focuses on finding paths that have overlapping stack frames with a target (uninitialized) variable.

Wang et al. present case studies about undefined behavior [46] among which are induced by the use of uninitialized variables. In a more recent work, Lee et al. introduce several methods to address undefined behavior in the LLVM compiler with a small performance overhead [25].

A popular attempt to tackle the problem of uninitialized memory reads is binary hardening. StackArmor [7] represents a hardening technique that is tailored to protect against stack-based vulnerabilities. To determine functions that might be prone to uninitialized reads, static analysis is used to identify stack locations which cannot proven to be safe. The system can protect against uninitialized reads but cannot detect them. SafeInit [30] extended this idea and represents a hardening technique specifically designed to mitigate uninitialized read vulnerabilities. The authors approach the problem from source code: based on Clang and LLVM, the general idea is to initialize all values on the

allocations site of heaps and stacks. In order to keep the overhead low, several strategies are applied to identify suitable spots. They modify the compiler to insert their initialization procedures. By leveraging a multi-variant execution approach, uninitialized reads can be detected. This, however, needs a corpus of proper inputs that can trigger those spots. UniSan [27] represents a similar approach to protect operating system kernels. Based on LLVM, the authors propose a compiler-based approach to eliminate information leaks caused by uninitialized data that utilizes dataflow analysis to trace execution paths that lead to possible leaking spots. UniSan checks for allocations to be fully initialized when they leave the kernel space, and instruments the kernel in order to initialize allocations with zeros, if the check is violated.

Another recent approach by Lu et al. [28] targets uninitialized reads in the Linux kernel. They propose techniques for stack spraying to enforce an overlap of the sprayed data with uninitialized memory. With a combination of symbolic execution and fuzzing, they present a deterministic way to find execution paths which prepare data that overlaps with data of a vulnerability.

Giuffrida et al. [19] present a monitoring infrastructure to detect different kinds of vulnerabilities, among them uninitialized reads. They perform static analysis at compile time to index program state invariants and identify typed memory objects. The invariants represent safety constraints which are instrumented as metadata into the final binary. Their approach also allows to update and manage metadata dynamically. The proposed framework monitors the application in realtime and checks for invariant violations. Ye et al. propose a static value-flow analysis [47]. They analyze the source and construct a value flow graph which serves to deduce a measure of definedness for variables. The analysis results are used to optimize the instrumentation process of binaries.

Other systems which instrument binaries either at compile or execution time to detect uninitialized reads at runtime are proposed in the literature [3, 42]. These systems can be used in combination with a fuzzer or a test suite to detect uninitialized variables. One advantage of these dynamic systems is that for each detected uninitialized bug an input vector can be derived. On the other hand, only executed paths will be detected and hence the code coverage is typically low. In addition, an appropriate corpus of input data is needed. In contrast, our static approach is capable of analyzing binary executables in a scalable way that provides high code coverage.

In summary, the wealth of work in recent research, most of which rely on source code, is tailored to instrumentation purposes to aid dynamic analysis in a monitoring environment. In contrast, our approach follows a purely large-scale static analysis that addresses the proactive detection of bugs in binary executables.

8 Conclusion

Uninitialized memory reads in an application can be utilized by an attacker for a variety of possibilities, the typical use case being an information leak that allows

an attacker to subsequently bypass information hiding schemes. In this paper, we proposed a novel static analysis approach to detect such vulnerabilities, with a focus on uninitialized stack variables. The modularity of our framework enables flexibility. We have built a prototype of the proposed approach that is capable of doing large-scale analyses to detect uninitialized memory reads in both synthetic examples as well as complex, real-world binaries. We believe that our system delivers new impulses to other researchers.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. This work was supported by the German Research Foundation (DFG) within the framework of the Excellence Strategy of the Federal Government and the States – EXC 2092 CaSa – 39078197. In addition, this work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (ERC Starting Grant No. 640110 (BASTION)).

References

1. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. Ph.D. thesis (1994)
2. Andriesse, D., Chen, X., van der Veen, V., Slowinska, A., Bos, H.: An in-depth analysis of disassembly on full-scale x86/x64 binaries. In: USENIX Security Symposium (2016)
3. Bruening, D., Zhao, Q.: Practical memory checking with dr. memory. In: Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (2011)
4. Budd, C.: Pwn2Own: Day 2 and Event Wrap-Up. http://blog.trendmicro.com/pwn2own-day-2-event-wrap/ (March 2016)
5. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering (1989)
6. Chen, X., Slowinska, A., Bos, H.: Who allocated my memory? detecting custom memory allocators in c binaries. In: 2013 20th Working Conference on Reverse Engineering (WCRE) (2013)
7. Chen, X., Slowinska, A., Andriesse, D., Bos, H., Giuffrida, C.: StackArmor: Comprehensive Protection from Stack-based Memory Error Vulnerabilities for Binaries. In: Symposium on Network and Distributed System Security (NDSS) (2015)
8. Chen, X., Slowinska, A., Bos, H.: On the detection of custom memory allocators in c binaries. Empirical Softw. Engg. **21**(3) (Jun 2016)
9. CVE-2012-1889: Vulnerability in microsoft xml core services. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2012-1889
10. CVE-2014-6355: Graphics component information disclosure vulnerability. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6355
11. CVE-2015-0061: Tiff processing information disclosure vulnerability. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0061

12. CVE-Statistics-Chrome: Google Chrome Vulnerability Statistics. http://www.cvedetails.com/product/15031/Google-Chrome.html (2014)
13. CVE-Statistics-Firefox: Mozilla Firefox Vulnerability Statistics. http://www.cvedetails.com/product/3264/Mozilla-Firefox.html (2014)
14. CVE-Statistics-IE: Microsoft Internet Explorer Vulnerability Statistics. http://www.cvedetails.com/product/9900/Microsoft-Internet-Explorer.html (2014)
15. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems (TOPLAS) **13**(4), 451–490 (1991)
16. Evans, I., Long, F., Otgonbaatar, U., Shrobe, H., Rinard, M., Okhravi, H., Sidiroglou-Douskos, S.: Control jujutsu: On the weaknesses of fine-grained control flow integrity. In: ACM Conference on Computer and Communications Security (CCS) (2015)
17. Fehnker, A., Huuck, R., Jayet, P., Lussenburg, M., Rauch, F.: Goanna a static model checker. In: Formal Methods: Applications and Technology, pp. 297–300. Springer (2006)
18. Flake, H.: Attacks on uninitialized local variables (2006)
19. Giuffrida, C., Cavallaro, L., Tanenbaum, A.S.: Practical automated vulnerability monitoring using program state invariants. In: Conference on Dependable Systems and Networks (DSN) (2013)
20. Haller, I., Slowinska, A., Bos, H.: MemPick: data structure detection in C/C++ binaries. In: Proceedings of the 20th Working Conference on Reverse Engineering (WCRE) (2013)
21. Hariri, A.A.: VMware Exploitation through Uninitialized Buffers. https://www.thezdi.com/blog/2018/3/1/vmware-exploitation-through-uninitialized-buffers (March 2018)
22. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis, vol. 20. ACM (1995)
23. Jin, W., Cohen, C., Gennari, J., Hines, C., Chaki, S., Gurfinkel, A., Havrilla, J., Narasimhan, P.: Recovering c++ objects from binaries using inter-procedural data-flow analysis. In: Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014 (2014)
24. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 553–568. Springer (2003)
25. Lee, J., Kim, Y., Song, Y., Hur, C.K., Das, S., Majnemer, D., Regehr, J., Lopes, N.P.: Taming Undefined Behavior in LLVM. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2017)
26. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.Y.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In Defense of Soundiness: A Manifesto. Commun. ACM **58**(2) (Jan 2015)
27. Lu, K., Song, C., Kim, T., Lee, W.: Unisan: Proactive kernel memory initialization to eliminate data leakages. In: ACM Conference on Computer and Communications Security (CCS) (2016)
28. Lu, K., Walter, M.T., Pfaff, D., Nürnberger, S., Lee, W., Backes, M.: Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In: Symposium on Network and Distributed System Security (NDSS) (2017)

29. Machiry, A., Spensky, C., Corina, J., Stephens, N., Kruegel, C., Vigna, G.: DR. CHECKER: A soundy analysis for linux kernel drivers. In: USENIX Security Symposium (2017)
30. Milburn, A., Bos, H., Giuffrida, C.: SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities. In: Symposium on Network and Distributed System Security (NDSS) (2017)
31. Molnar, D., Li, X.C., Wagner, D.A.: Dynamic test generation to find integer bugs in x86 binary linux programs. In: USENIX Security Symposium (2009)
32. Ramos, D.A., Engler, D.: Under-constrained symbolic execution: Correctness checking for real code. In: USENIX Security Symposium (2015)
33. Reps, T.: Undecidability of context-sensitive data-dependence analysis. ACM Trans. Program. Lang. Syst. **22**(1) (Jan 2000)
34. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on. pp. 55–74. IEEE (2002)
35. Richardson, S., Ganapathi, M.: Interprocedural analysis useless for code optimization. Tech. rep., Stanford, CA, USA (1987)
36. Rinetzky, N., Sagiv, M.: Interprocedural shape analysis for recursive programs. In: Compiler Construction. pp. 133–149. Springer (2001)
37. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. New York Univ. Comput. Sci. Dept., New York, NY (1978)
38. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: IEEE Symposium on Security and Privacy (2016)
39. Slowinska, A., Stancescu, T., Bos, H.: Howard: a dynamic excavator for reverse engineering data structures. In: Symposium on Network and Distributed System Security (NDSS). San Diego, CA (2011)
40. Smaragdakis, Y., Balatsouras, G.: Pointer analysis. Found. Trends Program. Lang. **2**(1) (Apr 2015)
41. Smaragdakis, Y., Bravenboer, M.: Using datalog for fast and easy program analysis. In: Proceedings of the First International Conference on Datalog Reloaded (2011)
42. Stepanov, E., Serebryany, K.: Memorysanitizer: Fast detector of uninitialized memory use in c++;. In: 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (2015)
43. Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: Eternal War in Memory. In: IEEE Symposium on Security and Privacy (2013)
44. Tillequin, A.: Amoco. `https://github.com/bdcht/amoco` (2016)
45. Van Emmerik, M.J.: Static single assignment for decompilation. Ph.D. thesis, The University of Queensland (2007)
46. Wang, X., Chen, H., Cheung, A., Jia, Z., Zeldovich, N., Kaashoek, M.F.: Undefined behavior: What happened to my code? In: Proceedings of the Asia-Pacific Workshop on Systems (2012)
47. Ye, D., Sui, Y., Xue, J.: Accelerating dynamic detection of uses of undefined values with static value-flow analysis. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (2014)