embeddedworld2021 Exhibition&Conference ...it's a smarter world DIGITAL

Increasing Resilience to Cyberattacks through Advanced Use of Static Code Analysis

Martin Becker
Application Engineering
The MathWorks GmbH
Ismaning, Germany
mbecker@mathworks.com

Jacob Palczynski
Training Services
The MathWorks GmbH
Aachen, Germany
jpalczyn@mathworks.com

***Abstract*—Embedded devices are suffering from an increasing number of cyberattacks across all industries and products. This trend continues although many developers are already using static code analysis in addition to dynamic testing. In this paper we identify possible reasons for this trend and propose techniques to address the underlying issues so that the attack resilience of embedded software can be increased.**

Specifically, we provide guidance on (1) which vulnerabilities can(not) be found by static code analysis, (2) how to control analysis context and setup to find more vulnerabilities, and (3) how to support root cause analysis and reduce false positives in library code. We also discuss how to anticipate unforeseen vulnerabilities in software and hardware.

Our findings are based on a study of over 60 CVEs from industrial and open source embedded software. Amongst them are the FreeRTOS vulnerabilities from 2018, of which approximately 80% could have been prevented with advanced use of static code analysis.

Keywords—security, static code analysis, security testing, vulnerability detection, formal verification

I. Introduction

Embedded software is ubiquitous in our daily lives. It adds convenience and intelligence to a plethora of systems ranging from headphones, over Wi-Fi routers, smartphones, and medical devices to autonomous cars, commercial aircraft, and military equipment. We put into its hands everything from our very personal secrets, to our lives and health. However, precisely because of the powerful role in these systems, they are also interesting targets for hackers. Malicious misuse, information disclosure, or causing their failure can result in financial damage, social manipulation, and sometimes even life hazard. The reality of such attacks is, unfortunately, visible in recent trends in cybersecurity [1], suggesting that attackers are very aware of the opportunities. Embedded software must therefore be designed to be resilient to cyberattacks if it shall be robust and trustworthy.

On the other hand, software engineers and developers have not stagnated either. A recent survey [2] has shown that more than 50% of open source projects make use of static code analysis tools to uncover defects and increase robustness, and our observation is that the prevalence is even higher in safety-critical and commercial software. Static code analysis (SCA) tools can verify the source code more thoroughly than dynamic testing [3] by performing a semantic analysis instead of executing test cases. In the context of security, they are referred to as *Static Application Security Testing* (SAST) tools. These tools are well-received by developers since they can point out shortcomings of designs as well as flawed coding styles [4]. Advanced SAST tools can precisely track data and control flows, and check hundreds of defect classes. Some are even able to *prove* the absence of errors with formal methods, which is equivalent to (the unattainable) exhaustive dynamic testing [5]. However, we observe that despite the increased use of SAST tools, the number of vulnerabilities reported for embedded software is still on the rise.

This paper explains when SAST may miss vulnerabilities and provides guidance on how to improve detection by advanced use such as tuning of analysis context, checker parametrization, contracts, and function replacements. We discuss why these and other methods help to uncover more defects, we link them to real-world vulnerabilities (CVEs), and we discuss which types of vulnerabilities evade tool capabilities. Moreover, we explain how developers can perform root cause analysis instead of just fixing symptoms. Part of this explanation is how to obtain partial attack paths and how to answer common questions; for example, Did I check all input data properly? Finally, we discuss the role and benefits of defensive coding and reduction of analysis assumptions. Both methods increase the resilience against unforeseen incidents, such as hardware vulnerabilities, but also bring positive side effects for SAST tools. The insights presented here summarize our forensic work on approximately 60 vulnerabilities in industrial and open source software with the *Polyspace Static Code Analysis Tools* [5]. To allow the reader to follow and replicate our methods, we provide examples of CVEs in open source software that are well-documented and already patched.

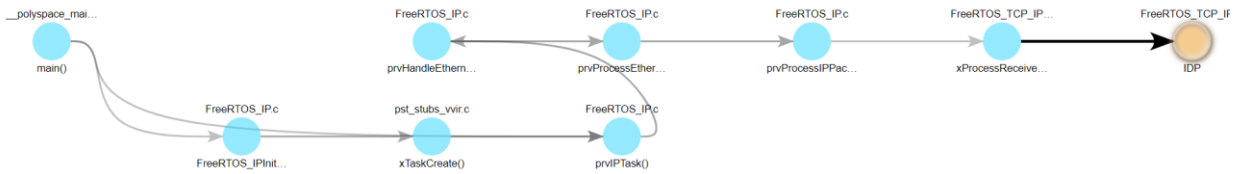

Figure 1. Error call graph for CVE-2018-16603 ("FreeRTOS TCP information leak") due to out-of-bounds read (CWE-125).

We start with the premise that coding guidelines like CERT-C have already been checked, since they can spot bad coding patterns that impair human comprehension and reduce the risk of missing defects. The Polyspace tools [6] can also do this job and notably increase software quality. However, checking and following coding guidelines is not sufficient to ensure resilience against cyberattacks, since there might still be dormant defects [4]. For example, there is no CERT-C rule that requires that the arguments of *memcpy* are consistent, which is however necessary to prevent, inter alia, memory corruption and information disclosure. More details are given in Section III.F. On the other hand, it may sometimes even be justified to violate coding guidelines, without necessarily having an impact on the security properties of the program. A trivial example is the cast of an integer into a narrower type, which is flagged by most coding standards, but may be a non-issue if the values of the wider type can be proven to be small enough. Advanced static code analysis can help to justify such violations by providing proofs that reduce review efforts. They also reveal vulnerabilities that have bypassed the guidelines. It is therefore advanced static code analysis that is the focus of this paper, and in particular tools based on *sound* formal methods, as explained later.

A. Related Work

A general overview about common methods in security testing and the important role of SAST tools is given in [3]. The authors rate SAST as an effective method for detecting programming-related vulnerabilities, with higher coverage and less false positives than dynamic, test-based methods (DAST).

Oyetoyan et al. [4] discuss common misconceptions of SAST tools, specifically in the context of agile workflows. They argue that SAST tools do not (magically) increase security since they focus on coding guidelines and metrics, and that those do not imply security. We agree with this argument for simple SAST tools, but we also demonstrate that advanced tools can go well beyond just compliance checking, and that proper use of such tools can significantly increase software security.

A study of vulnerability detection capabilities on the Juliet test suite has been published in [7]. The data shows that 27% of C/C++ vulnerabilities have been missed by all SAST tools, and that recall has been around only 50% (median). This is at the level of random guessing, as the authors point out. This paper picks up on this challenge by providing methods that increase the number of detected vulnerabilities (and thereby recall), and by providing methods that improve precision.

The authors in [8] and [9] are concerned with the difficulty of reviewing results of SAST tools, and therefore identified common questions that developers seek to answer during this process: Why is a warning shown on a particular line? What could be possible root causes? Which inputs may trigger this defect? This is the third aspect that we discuss in this paper.

II. Background: CyberSecurity at Code Level

A. Root Cause, Infection Chain, and Consequence

Let us begin with a precise definition of "defects" in context of security, following the terminology of Zeller [10]:

1. *Root cause*: an initial error created by a programmer
2. *Infection*: an error that was propagated by data- or control flow from either root cause or from a preceding infection
3. *Consequence*: the resulting unwanted behavior, such as program crash, remote code execution, etc.; this is often the focus of reported vulnerabilities

An *infection chain* is the event chain from the root cause to one or more infections and eventually to the consequence (Figure 1). Sometimes infection chains might cease and not lead to any consequences. In this paper, however, we focus on real vulnerabilities and therefore ignore this case together with non-accessible vulnerabilities (potential infection chains without a root cause). Every remaining infection chain is therefore exploitable and can be considered an *attack path*. Whenever the location of the root cause (and therefore the bugfix) is not known, we revert to the generic term *defect*.

B. Attack Scenarios, Common Weaknesses, and CVE

Cyberattacks have the goal of finding and exploiting consequences that occur at the end of an attack path. Usually, there are many different scenarios to launch an attack, depending on the window of opportunity, attack method, and the vulnerability that is thought to be dormant in the target system. However, one of the key elements in many attack scenarios is programming errors [11].

To address the well-known issue of programming errors, MITRE started the *Common Weakness Enumeration (CWE)* project in 1999. It captures and categorizes known programming errors to better understand common security flaws. The creation of CWE was part of the *Common Vulnerabilities and Exposures (CVE)* list, which was started to publish known vulnerabilities in a systematic way. Based on the CVEs of 2018 and 2019, MITRE has identified the top 25 most dangerous software weaknesses, with the top 10 most dangerous CWEs being [12]:

#	CWE	NV	CVSS	Name
1	*CWE-79	14%	5.8	Cross-Site Scripting
2	CWE-787	8%	8.31	Out-of-Bounds Write
3	CWE-20	7%	7.35	Improper Input Validation
4	CWE-125	6%	7.13	Out-of-Bounds Read
5	CWE-119	4%	8.08	Improper Restriction of Operations Within Bounds of a Memory Buffer
6	*CWE-89	3%	8.98	SQL Injection
7	CWE-200	5%	6.01	Exposure of Sensitive Information to an Unauthorized Actor
8	CWE-416	3%	8.26	Use After Free
9	*CWE-352	3%	8.08	Cross-Site Request Forgery (CSRF)
10	*CWE-78	3%	8.52	OS Command Injection

Whereas "NV" denotes the percentage of reported vulnerabilities referring to the CWE, and "CVSS" denotes the average of the official severity score.

As we can already see from this table, CWEs occasionally point to consequences (e.g., CWE-200) rather than to root causes (e.g., CWE-787). This is a known problem which is discussed in [13]. In this paper we use CWEs to classify root causes, unless noted otherwise.

The CVEs and methods shown in this paper address six out of 10 CWEs from this list. The remaining CWEs (marked *) have not been part of our forensics since they are atypical or rare applications for embedded systems, but we believe that the methods shown here can also be carried over to them.

C. On Static Code Analysis

Static code analysis tools should ideally not miss defects (no false negatives), but also not throw needless warnings (avoid false positives). This is, informally speaking, the essence of the famous Decision Problem in computer science, which states (with mathematical proof) that both *soundness* (no false negatives) and *completeness* (no false positives) are *impossible* to achieve simultaneously for meaningful formal systems [14], which also include C and C++ programs. Therefore, all SAST tools must either be unsound (miss defects) or incomplete (throw needless warnings). However, sophisticated tools throw only few warnings *and* miss few defects. A combination of both approaches is often useful in practice. Some tools even offer modes to choose between reporting only likely defects (low review effort, provides confidence for defect-free software) and reporting all defects (higher review effort, guarantees absence of defects in software). One such example is the *Polyspace Static Code Analysis Tools* [6].

In this paper we assume a *sound* static code analysis tool such as Polyspace Code Prover [5]. We argue that resilience to cyberattacks is closely related to robustness, which in turn depends on implementing all corner cases correctly. Thus, missing a defect can result in oversight of vulnerabilities. This, however, is not a concern with sound tools since they find as many potential defects as possible, and hence allow us to increase resilience through appropriate bugfixes. The methods shown here can also be applied with unsound tools, however, without any *guarantee* that the resilience to cyberattacks is indeed increased.

III. Methods to Uncover More Vulnerabilities

This section proposes best practices to avoid tool handling errors and to increase the sensitivity of SAST tools, both with the goal to reveal more vulnerabilities.

A. Remove Undefined Behavior First

Most vulnerabilities build on *undefined behavior* (e.g., 11 out of the 14 FreeRTOS vulnerabilities); that is, the unknown program behavior that results from operations which are not defined by the programming language standard and where consequently "anything can happen", including memory corruption and crash. Examples are dereferencing a NULL pointer or accessing an array beyond bounds. Therefore, most SAST tools aim to identify such bad operations.

However, undefined behavior is also the worst enemy of all SAST tools. Since, by definition, its presence means that the program behaves unpredictably, a sound tool has virtually no choice but to continue analysis under the assumption that the offending value or path is blocked from this point onwards.

These "blocking semantics" have two effects. First, they help identify problems closer to their root cause, which reduces debugging work. Second, they can also "hide" downstream defects in the code if a user chooses to ignore the tool warnings. Take the example of multiple out-of-bound array accesses in a row, caused by the same bad index variable: Only the first violation will be reported; thereafter the offending path is not further considered.

On a real target, undefined behavior can either agree with blocking semantics (e.g., program crash) or continue execution. The latter case, however, means the program lives on with different control flows than intended, and in an erroneous state. It is often this very effect that makes a vulnerability exploitable. Note that some SAST tools do not use blocking semantics. Instead, they only *flag* undefined behavior and continue analysis with the offending execution path, making the additional assumption that there is no other effect on program behavior. This, however, is not only unsound, but also throws more warnings since it considers execution paths which are infeasible as per the language semantics.

In summary, we urge developers to fix undefined behavior first, instead of suppressing or justifying such warnings from the tool. This can be done by proper setup of analysis context, as described next, or better even by defensive coding, as described in Section V.A.

B. Analysis Setup

Static analysis aims to predict the behavior of the program on the target processor. Towards this, advanced SAST tools, and especially sound ones, must take into account several aspects. Failure to consider these aspects results in either imprecise or, worse, unsound results.

Compiler and Target. The precise behavior of a program depends on the compiler, compiler settings, and processor. Therefore, a precise model of the target and toolchain is required to ensure that no defects are missed. Among others, the analysis must be set up correctly to consider:

- Target characteristics (such as endianness, rounding modes, word width, etc.),
- compiler- and implementation-defined behavior (such as alignment, pre-defined macros, intrinsics, and implicit types), and
- behavior of the standard library (such as different implementations of smart pointers between gcc and clang and their representation in memory).

Advanced SAST tools like Polyspace [5] therefore can automatically infer the correct settings by analyzing the build process (use of compilation databases, sniffing build commands). This not only ensures a correct analysis

setup and finds more vulnerabilities but also removes the burden of manual setup. In cases where build information is not available, users can still set up the analysis manually via a graphical user interface.

Scope and Context. Unless whole, self-contained programs (including main and library code) are analyzed, some function definitions will be missing. We discuss this in further detail in Section F for the standard library, where often only header files are available but not the implementation of functions such as *memcpy*. In such cases, SAST tools must work in the absence of function definitions ("stubbing") and make assumptions on their effects. Vice versa, if they encounter a function for which no caller is known, they must make assumptions on the calling context. By default, sound tools must make worst-case assumptions. Continuing the example of the unknown caller, they must assume that function parameters carry any value that is permitted by their type ("full range").

In some cases, users might have additional information available that can help SAST tools making better assumptions and thereby to obtain more precise results. Tools like Polyspace [5]allow to convey such information as *Data Range Specifications* (DRS). For example, it allows to define ranges on return values of stubbed functions, and to define ranges or pointer allocation properties of function parameters. However, we advocate using such mechanisms only when users have solid evidence that the assumptions are sound. Otherwise, vulnerabilities might be missed due to wrong user inputs.

Multitasking Model. Repeated and interrupted execution of functions may introduce additional program states and drastically change program behavior. For example, interrupts may manipulate global variables and/or enable and prohibit execution paths. Therefore, if multitasking is ignored, the analysis may draw incorrect conclusions about the data and control flow and eventually miss effects like overflows caused by repeated execution, illegal pointer assignments, and many other potential vulnerabilities caused by incorrect synchronization (deadlocks, partial writes). One example for this is CVE-2019-11922 ("zstd race condition", CWE-362). Advanced SAST tools like Polyspace [5] can automatically detect multitasking primitives of common targets, and furthermore allow parametrization to support custom board support packages and various operating systems.

Once target settings, analysis scope, and contextual assumptions have been chosen, we recommend running an initial analysis and to focus on dead code. Precise tools like Polyspace [5] can identify parts of the program that are not reachable with the given parameters. Since unreachable code cannot be executed as per the program semantics, it is not further analyzed. A problem occurs if the user-provided analysis assumptions are incorrect and lead to false dead code, since then vulnerabilities might be missed. Hence, users should first double check their analysis setup to ensure that whatever has been identified as dead code is indeed unused. If so, and only if the SAST tool is a sound one, the user might consider removing dead

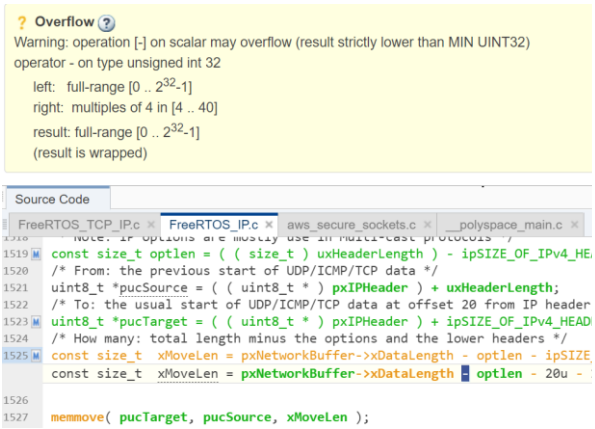

Figure 2. CVE-2018-16601 ("FreeRTOS IP DoS/Memory corruption") due to negative overflow (CWE-191). Enabling overflow checks on unsigned integers identifies the root cause.

code to increase performance. However, this is beyond the scope of this paper and should be done only after careful consideration, for it may counteract the principle of defensive coding (see Section V.A). Otherwise, the setup should be corrected accordingly (e.g., add preprocessor defines to make #ifdef contents reachable) to ensure that only truly unreachable code is classified as dead.

C. WYCIWYG (What You Check Is What You Get)

Another prerequisite for finding any given class of vulnerabilities is that the SAST tool actually implements the required checkers (and that the user activates them). Even if a SAST tool certifies a program to be free from defects, the program may still contain vulnerabilities. This is known by some practitioners [3] but is, unfortunately, a regular oversight by others. The following examples illustrate this "WYCIWYG" principle by example (noncomplete list):

- SAST tools which only check for undefined behavior (see Section A) cannot conclude the absence of DoS vulnerabilities caused by infinite loops or recursions.
- SAST tools which do not analyze multi-tasking or interrupts may fail to cover some execution paths in the program due to lacking variable interactions, and therefore miss vulnerabilities (see Section B).
- SAST tools which do not understand the semantics of crypto libraries cannot warn of predictable entropy sources as in CVE-2015-5611 ("Jeep Hack", CWE-337).
- SAST tools which do not understand the standard library can miss memory management issues, or cannot implement checks for TOCTOU vulnerabilities (see also Section F).

Many other capabilities could be mentioned here. Advanced SAST tools should therefore not only implement checkers for security coding standards like CERT-C but also take into consideration the mentioned aspects. The Polyspace family of static analysis tools [6] currently covers more than 280 defect classes, in addition to many coding standards like CERT-C(++) and MISRA-C(++).

D. Parametrization of Checkers and Tool Assumptions

Sophisticated SAST tools offer numerous parametrization options to allow fine-tuning of check behavior. Closely related to this, they also allow selecting default assumptions (see also Section B). Naturally, these options can decide whether a vulnerability is missed. Some examples are:

- Are environment pointers assumed unsafe? If yes, more defensive coding is required, but it also spots failure to check inputs (CWE-20).
- Are global variables considered as untainted? If yes, provide reasonable initialization values (CWE-454).
- Are overflows of unsigned types considered a defect? If yes, the root cause of CVE-2018-16601 can be spotted ("FreeRTOS DoS", CWE-191,, Figure 2).
- Are sub-normal floating-point numbers considered a defect? If yes, some timing side-channels like CVE-2017-5407 ("information disclosure", no CWE exists for root cause) can be spotted.

The Polyspace tools [6] offer the listed parameters among many others, and we recommend making use of them to detect a greater number of vulnerabilities.

E. Assertions and Contracts

Another effective method to catch more vulnerabilities is to encode assumptions or design intentions in the sources via assertions. Consider the example in Figure 4 from CVE-2019-13223 ("stb-ogg DoS", CWE-617): The developer expects that the variables *entries* and *dim* have a certain relation, and has captured this assumption as *assert(...)*. Dynamic testing could in principle exercise this claim. However, the offending state has not been found by DAST, resulting in this CVE. A similar case is CVE-2019-13219 ("stb-ogg DoS", CWE-476). A sound SAST tool like Polyspace [5], on the other hand, can prove or disprove this claim as shown, and warn the developer that this implementation deviates from the intent, and help to protect from this DoS vulnerability.

Moreover, assertions can be used to implement the well-known concept of contracts, like pre- and postconditions. For example, an assertion at the beginning of the function (precondition) can be used to capture allowed combinations of arguments. As an effect, SAST can spot incorrect use of a function. Vice versa, and like in this example, we can also document the intended state on completion of a function (postcondition), which enables SAST to prove or disprove that the implementation of the function itself is following intentions.

```
1200 static int lookup1_values(int entries, int dim)
1201 {
1202    int r = (int) floor(exp((float) log((float) entries) / dim));
1203    if ((int) floor(pow((float) r+1, dim)) <= entries)   // (int)
1204       ++r;                                              // floor
1205    assert(pow((float) r+1, dim) > entries);
1206    assert((int) floor(pow((float)
1207    return r;
1208 }
1209
```

operator > on type float 64
left: [0.0 .. 1.7977E^+308]
right: integer values in [1.0 .. 2.1475E

Figure 4. CVE-2019-13223 ("stb-ogg DoS") due to failing assert (CWE-617) predicted by SAST, including offending data. Assertions and contracts are effective for encoding assumptions, but also potential vulnerabilities.

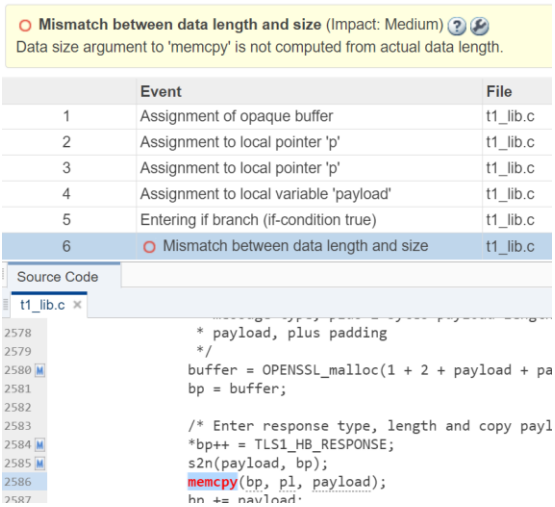

Figure 3. CVE-2014-0160 ("SSL heartbleed", CWE-119) with complete attack path.

Arguably, in some cases it might be better to sacrifice well-defined behavior for availability of service. In such a case, it can be justified to turn off assertions for the deployed system, yet still benefit from assertions with SAST.

F. Customizing and Mapping of stdlib Replacements

By default, sound SAST tools must perform stubbing when not all source code is available, as discussed in Section B. However, stubbing can lead to oversight of vulnerabilities, too. Consider the function *pvPortMalloc* of FreeRTOS, which is used similarly to *malloc* from the C standard library. If a SAST tool does not "know" its meaning, it a) cannot conclude that a new memory region is allocated nor the size thereof, b) cannot warn of problems with dynamic memory management, such as error checking and initialization problems, and c) cannot track out-of-bounds memory access errors.

Another example is that developers might want to re-implement a target-specific math function as a replacement for a slower standard library function. SAST tools can only check for domain errors (like passing negative value to a *sqrt* replacement) if this equivalence is known. If there is a one-to-one mapping from the original function signature to the custom one, an appropriate macro definition might be enough (but still must be provided). In contrast, if arguments are switched or added, a mapping between the custom function and the replaced library function must be established by other means. In the case of Polyspace, an XML file conveys such relations.

Mapping and replacement capabilities are especially important to detect vulnerabilities around memory management. In the case of the FreeRTOS vulnerabilities, failure to consider replacements would have caused oversight of seven out of the 14 CVEs.

G. What Do We Still Miss?

In extension to the WYCIWYG principle, some vulnerabilities are essentially invisible to SAST tools, regardless of setup, implemented checkers, or tool precision. It is important to understand this limitation so that security experts can focus their efforts on exercising such cases with

other methods, like penetration testing. Under the following circumstances, vulnerabilities cannot be detected by a SAST tool (non-exhaustive list):

Incorrect functional properties. The DNS poisoning of vulnerability of FreeRTOS (CVE-2018-16598, CWE-441) was defect-free from the tool's point of view. However, the *logic* deciding which DNS queries should be stored in cache did allow the eviction of useful cache contents for garbage data. As an effect, the cache became practically useless, and system performance dropped below acceptable limits. The consequence is Denial of Service. Clearly, such defects where the knowledge of correctness is beyond the source code are usually impossible to detect for SAST tools. Similar undetectable cases are incorrect formation of network packets where length fields are inconsistent with payload length (CVE-2018-16527, "FreeRTOS leak", conseq. CWE-200) or, in general, any implementation that deviates from a specification that is not part of the source code.

Temporal properties. One type of vulnerability is timing side-channels. Since SAST tools do not perform timing analysis, such defects usually cannot be detected. An exception is CVE-2017-5407 ("information disclosure"), as discussed in Section D.

Insufficient analysis scope. Consider the example of a program forming a NULL pointer. Since this is not a defect, SAST tools will not report it. However, if this pointer is later dereferenced in another part of the program that is not part of the analysis, a DoS vulnerability has been missed. This demonstrates a common challenge with library code like FreeRTOS (CVE-2018-16522, "DoS/RCE", CWE-665) and is another reason to use contracts (see Section E) to ensure the library is implemented correctly.

All the above limitations also apply to DAST tools; therefore, such vulnerabilities may go undetected. Nevertheless, they shall be addressed. We have already discussed how assertions and contracts can be a remedy, alleviating some of these fundamental limitations. Unfortunately, there are practical limits to what can be expressed in contracts. For example, temporal properties are mostly beyond the scope of assertions. We therefore recommend identifying all inexpressible security properties as part of the asset and threat analysis or the cybersecurity concept, where such claims usually have to be documented and verified by other means like penetration testing [15].

IV. Methods to Support Root Cause Analysis

As we have seen, findings may point to consequences, rather than to root causes. This can not only make it tough to comprehend tool warnings, but in some cases they can be confounded with false positives and subsequently ignored, leading to oversight of vulnerabilities [8]. What developers ultimately need is to see the root cause of identified vulnerabilities to answer their most frequent questions: Is it a real issue? How can it happen? How can I fix it? [9]. In this section, we show how to help developers better understand findings, assess attack packs, and localize design errors before infection chains propagate through the program.

A. Leverage Tool Capabilities

The perhaps most useful method for root cause analysis is to use SAST tools that can provide contextual information for failed checks; for example (non-complete list):

- *Call context*: Which caller provides offending input that cause a vulnerability (context sensitivity)?
- *Control flow*: Which decisions have been taken to get here (path sensitivity)?
- *Variable values*: What are possible values of array-indexing variables (CWE-787, CWE-125)? Are variables always initialized before use (CWE-665)?
- *Pointer analysis*: Are they initialized? Where do they point to, and how large are the underlying memory regions (CWE-119)?
- *Taint analysis*: Is a function using potentially malign user inputs without prior sanity checking (CWE-20)?
- *Accesses of globals*: Where are they used and how (read/write)? Are there possible race conditions?

More sophisticated SAST tools like Polyspace [6] compute *event traces*, which summarize all this information in a sequence of steps. Users can interactively walk through the sequence, inspect variables and pointers, track control flow decisions (see Figure 3 and Figure 5), and thereby truly comprehend why a tool warning is emitted for a specific operation in the code. Referring to our initial nomenclature, event traces are a partial representation of the infection chain, and therefore, an attack path can often be derived from them without much effort.

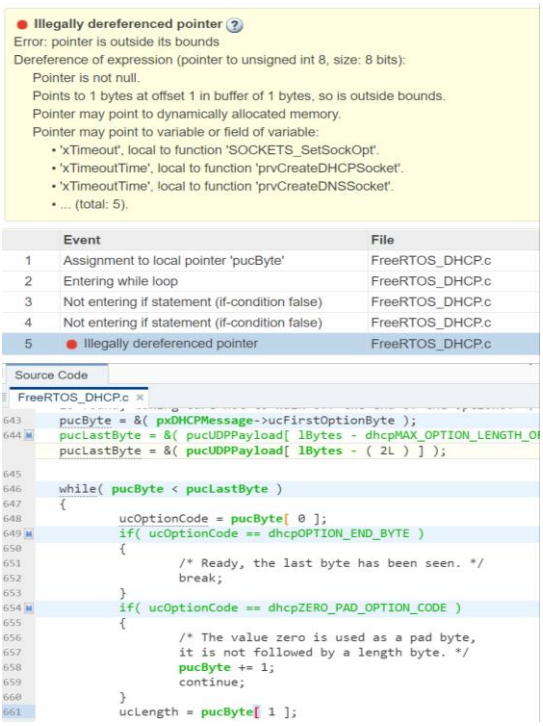

Figure 5. CVE-2018-16602 ("FreeRTOS DHCP leak", CWE-125) with partial attack path.

However, we observe that such capabilities of SAST tools are often ignored in automated CI/CD setups, forcing developers to invest an unreasonable amount of time to reconstruct information from logfiles or general-purpose dashboards. Therefore, we recommend exposing review interfaces of SAST tools to users for results drill-down, and to consider replacing SAST tools that do not have interactive capabilities. This helps addressing typical questions during results review, and often can dramatically speed up the debugging and resolution process.

B. Contracts and Assertions

As shown before, contracts and assertions are an effective way to spot deviations from the intended behavior. However, they also simplify root cause analysis by acting as barriers for the silent propagation of infection chains. Additionally, assertions are useful to "test" knowledge of the SAST tool which may not be displayed explicitly.

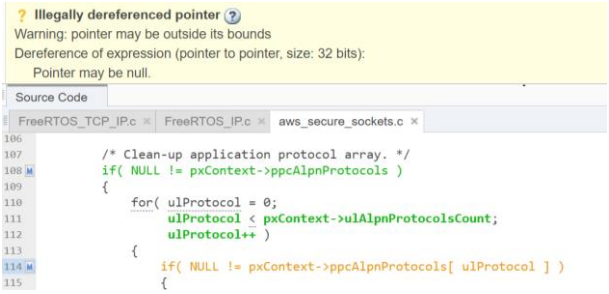

Figure 6. CVE-2018-16522 ("FreeRTOS DoS/RCE") due to improper initialization (CWE-665) at the location of consequence. A bugfix is not possible here. Contracts identify the root cause.

As an example, consider CVE-2018-16522 ("FreeRTOS DoS/RCE", CWE-665) shown in Figure 6: This is the *SOCKETS_Close* function of FreeRTOS, which is called by userland code. Therefore, the contents of *pxContext* could be anything. The analysis has found that the dereference in line 114 is potentially unsafe, but there is no way to fix it at this code location. It might be justified to provide contextual assumptions, as discussed in Section III.B. However, we must ensure that such assumptions are indeed true; that is, analyze whether the code that sets up the context is indeed guaranteeing them. Towards this, a postcondition for *SOCKETS_SetSockOpt* would be the appropriate method and spot the problem at the root cause, rather than the late and unfixable consequence:

```
#ifdef POLYSPACE
for (int i=0; i<NPROTO; ++i)
    assert(xSocket->ppcAlpnProtocols[i]);
#endif
```

This is a SAST-only contract that includes an "is-initialized" check when the *i*-th array element is accessed. The marked index operator is indeed faulty here (highlighted orange in Polyspace), indicating that on return of *SOCKET_SetSockOpt* initialization may be incomplete.

As mentioned before, contracts and assertions have the additional beneficial that they can be exercised through dynamic testing. This particular CVE, however, is difficult to trigger since both the number of protocols must exceed one and the attacker must exhaust the memory on the target system. This is a good example that showing the superior effectiveness of SAST over DAST mentioned in [3].

V. Increasing Resilience to Unknowns

With the methods discussed so far, 11 out of 14 FreeRTOS CVEs could be found. This last section proposes two aggressive methods that can increase resilience further to maintain an operational system even if some potential hardware or software vulnerabilities remain undetected.

A. Defensive Coding

Defensive programming or coding practices explicitly take into consideration unforeseen circumstances as means to increase robustness and resilience of the software. In practice, this is done by extensive error checking of program states, call parameters, and return values. Note that this might have a negative impact on performance and may be identified as dead code (see Section III.B), which is, however, not the focus of this paper.

Defensive coding brings two benefits for our goal of increasing resilience: (1) As opposed to contracts, which may still be violated, defensive coding introduces real robustness. Infection chains cease through defensive coding, which leads to higher resilience even if analysis assumptions are broken during operation. This is, for example, the case when parts of the system are compromised by a hacker. Note that compiler flags might be necessary to avoid optimizers from removing such code. (2) Defensive code acts as a "filter" for SAST tools and improves precision. Since sound tools must consider all program states permissible by the analysis context, they often yield more warnings than developers may agree with (remark: often unrightfully so). Precise SAST tools, however, benefit from defensive coding by recognizing that offensive values do not reach critical parts.

As tip for readers who utilize Model-Based Design to generate code from graphically designed algorithms, we like to point out that advanced code generators are able to generate defensive code by merely setting options correctly, as is the case with *Embedded Coder* [16].

B. Minimize Contextual Assumptions

Precisely providing analysis assumptions on the calling context may result in fewer warnings (see Section III.B), but might also open the door to attacks if assumptions are violated, as discussed earlier. If a hacker can manipulate one part of a program such that assumptions are violated for another part, a chain effect may occur after which the program follows execution paths that have never been analyzed, wreaking havoc (e.g., through a *NOP slide*).

Therefore, we argue that for resilience, assumptions should be avoided as far as possible already while the program is still under development. This reinforces the need for defensive coding and hence fosters resilience. For an already developed software, assumptions should be gradually removed; e.g.:

- Omitting contextual code (such as test harnesses or main functions) for analysis,
- enforcing of stubbing for individual functions, or
- removing range constraints or information about global variables.

Naturally, this results in more findings and hence reveals potential vulnerabilities that have been unreachable before. We again discourage simply suppressing them, since in our experience this is a major reason for missed vulnerabilities. Instead, fixes should be applied until SAST is able to prove freedom from (checked) defects.

Moreover, the reduction of contextual assumptions brings two immediate benefits for SAST tools:

1. *Speedup of analysis*: In absence of contextual assumptions, analysis can be performed on smaller program partitions without further loss of precision, which reduces analysis time.
2. *Increase of precision*: Smaller program partitions imply less complexity, which allows some tools to omit otherwise necessary overapproximation. Moreover, precision is further increased by the "filtering effect", as explained earlier.

With Polyspace [5], the option "-unit-by-unit" has the effect of ignoring cross-unit analysis context and thus will reap the benefits listed above. Analysis context can be further reduced by avoiding or undoing what was discussed in Section III.B. The result will be a software that is resilient against unforeseen vulnerabilities in the system, which, as we have seen in recent history, can also stem from hardware. The presented methods to increase resilience should therefore not be considered optional, but rather as means to anticipate of the imperfection in today's complex embedded systems.

VI. Concluding Remarks

We have discussed how SAST tools can spot many vulnerabilities and how their effectiveness can be improved through advanced use. First, undefined behavior should be addressed, as it can hide vulnerabilities. Next, the analysis setup and tool configuration should be chosen carefully. We have covered aspects of scope and contextual assumptions (and their dangers), as well as checker configuration and function replacements. With this, SAST can detect more than DAST in many cases, confirming its important role for cybersecurity.

However, there are vulnerabilities which neither DAST nor SAST can detect, namely those requiring knowledge or context beyond the sources (e.g., the meaning of data fields in network packets or incorrect logic of algorithms), those that have their consequences beyond analysis scope (e.g., analyzing a library but not its correct use), as well as vulnerabilities related to timing properties.

Proper root cause analysis of the tool's findings is another important aspect, since otherwise findings might be unrightfully suppressed, leading to missed vulnerabilities, too. We have shown how sophisticated SAST tools support this process and advocate that developers should get access to appropriate review interfaces.

Finally, we have proposed methods to handle unforeseen circumstances, namely defensive coding as well as contracts, and explained how they can benefit analysis precision, reduce the analysis time, and further increase the resilience to cyberattacks.

VII. References

[1] The MITRE Corporation, "CVE Vulnerabilities by Date," 2020. [Online]. Available: https://www.cvedetails.com/browse-by-date.php. [Accessed 12 12 2020].

[2] M. Beller, R. Bholanath, S. McIntosh and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *International Conference on Software Analysis, Evolution, and Reengineering*, 2016.

[3] M. Felderer, M. Büchler, M. Johns, A. Brucker, R. Breu and A. Pretschner, "Security testing: A survey," *Advances in Computer Science,* vol. Vol. 101, pp. pp. 1-51., 2016.

[4] T. Oyetoyan, B. Milosheka, M. Grini and D. Cruzes, "Myths and Facts About Static Application Security Testing Tools," *Lecture Notes in Business Information Processing,* vol. Vol. 314, 2018.

[5] The MathWorks Inc., "Polyspace Code Prover," 10 December 2020. [Online]. Available: https://www.polyspace.com.

[6] "Polyspace Static Code Analysis Products," The MathWorks Inc., 2020. [Online]. Available: https://www.polyspace.com. [Accessed 16 12 2020].

[7] K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," *Information and Software Technology,* vol. Vol. 68, 2015.

[8] L. Ben Othmane, P. Angin, H. Weffers and B. Bhargava, "Extending the agile development process to develop acceptably secure software," *IEEE Transactions on Dependable and Secure Computing,* vol. Vol. 11, no. 6, pp. 497-509, 2014.

[9] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu and H. Lipford, "Questions developers ask while diagnosing potential security vulnerabilities with static analysis," in *Proceedings of Joint Meeting on Foundations of Software Engineering*, 2015.

[10] A. Zeller, Why Programs Fail, Elsevier, 2005.

[11] D. Papp, Z. Ma and L. Buttyan, "Embedded systems security: Threats, vulnerabilities," in *13th Annual Conference on Privacy, Security and Trust (PST)*, 2015.

[12] The MITRE Corporation, "2020 CWE Top 25 Most Dangerous Software Weaknesses," 2020. [Online]. Available: https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html. [Accessed 14 December 2020].

[13] The MITRE Corporation, "CWE Mapping Analysis," September 2008. [Online]. Available: https://cwe.mitre.org/documents/mapping_analysis/index.html. [Accessed 15 December 2020].

[14] A. Turing, "On computable numbers, with an application to the Entscheidungsproblem," in *Proceedings of the London mathematical society*, 1937.

[15] SAE International, *ISO/SAE DIS 21434 Road Vehicles Cybersecurity Engineering,* 2020.

[16] The MathWorks Inc., "Embedded Coder," The MathWorks Inc., December 2020. [Online]. Available: https://www.mathworks.com/products/embedded-coder.html.

[17] F. Caron, "Obtaining reasonable assurance on cyber resilience," *Managerial Auditing Journal,* 2019.