The NSoE-TSS Technology Consolidation Review 2021

The National Satellite of Excellence in Trustworthy Software Systems
(NSoE-TSS)

Table of Contents

Background 3

Fuzz Testing 5

- How does fuzzing help? 5
- What are the challenges in fuzzing? 6
- SWOT Analysis for Fuzzing 10

Program Repair and Binary Hardening 11

- How does binary hardening and repair help? 11
- What are the challenges in binary hardening and repair? 12
- What are the risks with binary hardening and repair? 15
- SWOT Analysis for Binary Analysis 16

Artificial Intelligence Testing 17

- How does AI quality assurance techniques help? 17
- What are the challenges in building trustworthy AI systems? 18
- SWOT Analysis for AI quality assurance 22

Formal Verification 23

- How can formal verification help? 23
- What are the challenges in formal verification? 24
- SWOT Analysis for Formal Verification 26

Bibliography 27

Background

Singapore is one of the most digitally connected nations of the world, and is well poised to make a broad bold move towards building full-fledged smart nation infrastructures. Singapore has acquired all the necessary ingredients for building a smart nation including full digital penetration, connectivity, mobile infrastructures and autonomous systems. This has been achieved through research, innovation, enterprise, and planning. Against this backdrop, there exist two principal challenges today, namely standardization of different public and private services for common tasks, and increased adoption of integrated smart systems.

The *National Satellite of Excellence in Trustworthy Software Systems* (NSoE-TSS) aims to enhance Singapore's national capabilities in trustworthy smart system infrastructures. We seek to build on our combined strengths in software security, and smart systems to build an array of point technologies, leading to certification capabilities of embedded software systems. The certification can take on a range of flavors including functionality certification, checking against crashes and vulnerabilities, measuring and certifying resilience against malicious inputs and environments, as well as checking and certifying for absence of information leakage via extra-functional mechanisms such as side channels.

Since inception in 2019, the NSoE-TSS has conducted cutting-edge research into software and system security. The center aims to build on Singapore's combined strengths in the areas of analysis, testing, verification, hardening, isolation and system design. The focus of the NSoE-TSS is the development certification capabilities, and providing the tools to enhance or verify the trustworthiness of modern software systems.

In the context of the certification of software systems, the NSoE-TSS has decided to focus on four main core capabilities, namely:

- *Fuzz testing*: automatically discovering bugs and security vulnerabilities in software systems
- *Binary analysis, hardening and repair*: detecting and repairing bugs in software systems without access to source code
- *Artificial Intelligence*: towards testing and verifying the trustworthiness of emerging technologies, such as deep neural networks; and
- *Verification*: towards the verification of software systems with strong guarantees.

Each of these complementary capabilities sits at the forefront of modern security research, and provides the most potential for real-world impact. For example, fuzz testing has proven to be a very effective and practical method for the detection of bugs and other vulnerabilities in real-world software, as well as being an active area for security research. Research discoveries in the area of fuzzing can very likely be applied in both government and industry applications.

Binaries analysis with hardening and repair is another important core capability. In the real-world, much of the software for critical infrastructure and other software systems is close-

sourced, meaning that the source code is not available. This also means that these software components cannot be inspected in order to detect bugs or other issues. One solution is binary analysis, which attempts to analyze the binary code directly without any access to source code. Furthermore, the NSoE-TSS is developing technologies for direct binary hardening and repair, without cooperation with the original software developer.

Artificial Intelligence (AI) systems have gained much popularity in real-world applications. Like the traditional software, it is also important to guarantee the trustworthiness of AI systems, especially for those that are safety- and security-critical. The research on determining the trustworthiness of AI systems remains at an early stage, due to the inherent uncertainty of the systems' behaviors and outputs. The techniques in traditional software are difficult to be applied in the new domain. The NSoE-TSS aims to develop technologies for evaluating and enhancing the trustworthiness of AI systems including runtime monitor, repair, noisy label handling and adversarial example detection.

On critical and high-assurance systems, it is essential to have strong guarantees on the functional and security correctness of the system. For complex systems like concurrent and distributed software, where traditional techniques cannot cover the complete state space of an application, formal methods, and in particular formal verification, have been successfully used in proving that a system fulfills its functional and security specification. NSoE-TSS develops technologies and theories to progress in the state of the art of techniques for concurrent systems that are escalable for the formal verification of real world systems.

The rest of this document is organized into the four main core capabilities of the NSoE-TSS, where the motivation, use-cases and future challenges are discussed in more detail. For each core capability, we also include a SWOT analysis to highlight potential Strengths, *Weaknesses*, *Opportunities* and *Threats* at-a-glance. The aim of this document is to assist in future planning, especially in relation to investment of resources into future research projects, or to bring existing research projects into practical use.

Fuzz Testing

Fuzzing has been proven to be an effective method for software testing, and is responsible for the detection of the majority of security vulnerabilities in modern software systems. At its core, fuzzing is a biased random search over program inputs, with the goal of uncovering inputs that cause a crash, hang, or some other observable problem. Fuzzing has shown to be much more effective than manual or human-based testing, since the typical fuzzing tool can automatically generate hundreds or thousands of test cases quickly. Feedback from the program, such as path coverage information, can also guide test generation, making it more likely for new test cases to explore new paths/code in order to uncover problems.

Fuzzing is currently a hot topic in the software engineering and cybersecurity research communities. This has led to the rapid development of several technologies, tools and prototypes, each with the aim of improving fuzzing performance, effectiveness, or applicability to new software domains (such as Network programming, etc.). Some of these tools have been developed as part of the NSoE-TSS. Given the potential real-world impact of improved software testing, we anticipate this research trend will continue over the next decade.

Fuzz testing is a core competency of the NSoE-TSS project. While existing fuzzing work focuses on traditional software domains (e.g. Linux programs), we aim to extend fuzz testing into new areas, such as Deep Neural Networks (DNNs) and compilers.

How does fuzzing help?

Fuzz testing is a method for automated software testing. Fuzz testing differs from traditional software testing. With the latter, the developer (or testers) will curate a test suite in order to provide some level of software quality assurance and detect bugs. Test cases may be created either proactively (e.g., developer provided) or retroactively (e.g., user bug reports). Test cases may also be created with respect to some metric such as code coverage, i.e., how much of the code is executed by the test suite? Software engineering experience has proven that software testing is an essential part of the software development process.

Unlike traditional software testing methods, fuzz testing aims to be fully automated, and is based on some form of (guided) random search. Instead of a fixed test suite, fuzz testing will automatically generate new test cases by applying random mutations to existing tests, allowing for hundreds or thousands of tests per second. Most fuzz testing tools rely on feedback from the program in order to guide the search. The feedback can be in the form of code coverage information (greybox fuzzing) or symbolic information (whitebox fuzzing). Overall, greybox fuzz testing has shown to be very effective at testing software, since the volume of tests can often uncover even rare or hard-to-find bugs.

Fuzz testing can usually only detect specific classes of bug, such as software crash or assertion failure. Fuzz testing usually cannot detect logical bugs such as incorrect program output, since these do not have an observable outcome (e.g., a crash) that can be easily detected. Despite this, the kinds of bugs that fuzz testing can detect are usually security critical, since a crash usually corresponds to a denial-of-service opportunity, or may indicate an underlying exploitable error, such as a buffer overflow or use-after-free error. This has made fuzz testing very impactful, and is the main source of newly discovered security bugs or vulnerabilities discovered in modern software systems.

What are the challenges in fuzzing?

There are several challenges in the translation of fuzz testing technologies.

How can we promote developer awareness and adoption?

The topic of fuzz testing has received increased attention over the last decade, with popular fuzz testing tools, such as the *American Fuzzy Lop* (AFL), being first developed in 2013. As such, fuzz testing may still be seen as relatively "new" technology among some software developers, where there may be a residual preference for more traditional software testing methodologies. Although we expect this to slowly improve over time, the promotion of fuzz testing technologies is important.

Possible Solutions: One way to help promote fuzz testing technologies would be to create standards or certification based on best fuzz testing practice. For example, before software is released, the software has been subject to a fuzz campaign of some specified resource such as time or coverage. This can be integrated into, or in addition to, existing software testing infrastructure. Building the technologies suitable for the development of such standards is one of the aims of the NSoE-TSS.

Another idea is to provide developers with the resources needed to make the successful application of fuzzing possible. One idea would be *fuzzing-as-a-service*, where the resources for fuzzing are provided by a third party. An example model would be Google's OSS Fuzz infrastructure, where open source projects can upload their projects to be fuzz tested using Google's servers, which allows for significant computational power to be used beyond that available to individual projects. This project has also proven to be very successful, with countless bugs and vulnerabilities discovered in open-source software. To replicate this for commercial or close-sourced software systems would require a trusted third party, such as a government agency, to provide the infrastructure.

How can we fuzz more types of software systems?

Another significant challenge is how to fuzz arbitrary software? Some common fuzz testing tools, such as AFL, are designed for Linux binaries by default, and a port is needed to fuzz

software on other operating systems, such as winAFL for Windows. However, these ports may be less effective in terms of fuzzing speed and/or may have usability issues.

The problem can be more complicated in relation to embedded or IoT software. There are several challenges, such as the diverse nature of IoT software and hardware, diverse operating systems, and running on diverse system architecture, that must be addressed. A solution designed for a specific IoT device cannot readily be adapted to a different environment. For example, many home routers (supplied by ISPs) use a variant of MIPS Linux, whereas other low-power smart devices use an embedded real-time operating system running on ARM Thumb. There is no meaningful generalization between such systems, and each will require a specialized fuzzing solution. The amount of work required can be prohibitive, which means that many smart devices are not fuzz tested.

Other application domains include fuzzing for binary repair or deep neural networks. As with IoT, these tend to be specialized domains with specialized fuzz testing solutions.

Possible Solutions: There may be a trend towards generalizing fuzz testing technologies, making it possible to apply the approach to new domains. However, it is also likely there is no silver bullet, meaning adapting fuzzing technologies into new domains still requires manpower investment. This is likely unavoidable in the general case.

One example domain is IoT and smart devices, where each device is different and is essentially a sub-domain in its own right. The problem of fuzzing individual devices can be partly solved by software *emulation*, meaning that the software is run in a simulated environment. However, accurately simulating systems-level software is well known to be difficult in the general case, with a significant investment required in order to emulate the hardware/environment realistically. Alternatively, fuzz testing may be performed on the actual device, thereby avoiding some of the complexities and limitations of emulation. However, this may introduce new problems, such as low power devices being unsuitable for fuzzing, or the problem of uploading a fuzz testing tool onto the device itself.

The IoT domain is just one example. The NSoE-TSS has also completed research that has brought fuzz testing into novel domains, including: (1) *VulnLoc* [VulnLoc], a tool for localizing vulnerabilities using concentrated fuzzing, (2) *OASIS* [OASIS], a hybrid kernel symbolic execution framework for malware analysis, (3) *Sensei* [Sensei] and *DeepHunter* [DeepHunter] for repurposing fuzzing techniques to augment Deep Neural Network (DNN) training data, (4) Fix2Fit [Fix2Fit] fuzzing for binary repair, (5) Concolic Program Repair [CPR], and (6) TimeMachine [TimeMachine] testing for Android apps. Each of these projects was a specialized effort, where the concept of fuzz testing was adapted into a very specific problem domain, often with encouraging results. This highlights the potential for further expansion along these lines and we expect this research trend to continue as future work.

How can we improve the usability of fuzzing tools?

Another significant adoption barrier is the usability of fuzz testing tools. As a result of the recent explosion into fuzz testing research, many prototype research tools have been developed, including several projects from the NSoE-TSS. Each prototype typically targets some specific improvement or domain. This has led to a somewhat fragmented landscape, where each incremental improvement exists only as a specific research prototype (e.g., see the various fuzzing or AFL extensions published on Github), where usability is not the main priority. Other fuzzing technologies, such as whitefox fuzzing based on symbolic execution, have scalability limitations making it unusable for most real-world software. Furthermore, there is no "grand" integration of the existing state-of-the-art fuzzing research, where each new project can be integrated into an existing framework.

Another problem is that each individual tool may also have its own usability issues. A typical tool requires the project to be built using a modified compiler, and this may be difficult to integrate into existing build systems for complex projects. For example, the *TSuNAMi project* from NUS published the *AFLGo tool* for directed fuzzing on Github. However, running the tool requires no-less than 9 steps, including setting up and building other tools as dependencies.

Even standard fuzzing tools have usability issues from the developer perspective. For example, standard AFL requires the program to be recompiled using a specially modified compiler, which must be integrated into the project's existing build system. For large projects this can be a non-trivial task. Also, changing compilers can introduce compatibility problems, possibly leading to false detections of errors which further frustrate the testing process. There is also the possibility that the fuzz testing process causes unwanted side effects on the host system, such as the creation of unwanted files, which must be mitigated by the developer.

Possible Solutions: In the context of practical greybox fuzzing, there are third party efforts to integrate incremental improvements to fuzz testing under a unified framework, such as AFLPlusPlus. However, given the significant body of research work related to fuzzing, even AFLPlusPlus is selective. For example AFLPlusPlus supports normal greybox-style fuzz testing but not other research variants, such as directed fuzzing and AFLGo mentioned above.

Recent binary-only fuzzing solutions offer another partial solution to the tool usability problem for standard fuzzing workflows. One example is the E9AFL tool [E9AFL], recently developed by the NSoE-TSS, which allows binary programs to be fuzzed without recompilation using a modified compiler. This allows the problem to be compiled as normal and fuzzed directly, which has the advantage of not interrupting the developer workflow by requiring modifying the build system. That said, binary-only methods are traditionally viewed as being less reliable, slow, or buggy compared to source-only recompilation. However, we believe this is no longer the case with modern binary-only tools, such as E9AFL, which can successfully scale to very large programs.

How can we assess residual security risk if the fuzzing campaign was unsuccessful?

Technologies such as coverage guided greybox fuzz testing are imperfect, since 100% coverage of real-world programs is rarely achievable. Unguided blackbox achieves lower coverage since there is no feedback to guide the process to explore new paths. Whitebox fuzzing solutions also suffer from practical difficulties in terms of poor scalability, meaning that only a small part of the program can be properly tested. These issues mean that only some fraction of the program can be reached by fuzz testing, and the rest of the program will remain untested. These untested paths may harbor bugs that are never reached by the fuzzer. Even for the reached paths of the program, there is no guarantee that all bugs have been discovered, since some bugs may be dependent on data flow (i.e., what was the program input?) in addition to control-flow covered by the fuzzer.

As such, fuzzing provides no guarantee that the subject program is free from errors for any given time budget. For security critical applications, the residual risk may still be unacceptable, even after significant resources have already been devoted to fuzz testing. This is a well known problem that affects fuzzing and any other software testing methodology in general.

Possible Solutions: Usually, some trade-off between testing effectiveness and practicality is unavoidable. One idea is to mitigate the risks of imperfect fuzz testing with some other complementary approach. For example, the combination of fuzz testing with hardening (such as CFI and memory error hardening) may be a reasonable practical compromise. In other words: fuzzing will try to detect as many errors as possible, so that the errors can be fixed before shipping. For any remaining (undiscovered) bug, we rely on hardening.

Another approach would be to throw more resourcing into the fuzzing process, with the hope that more difficult-to-reach bugs will be discovered. This can also be another argument for fuzzing-as-a-service, as illustrated by projects such as Google's OSS Fuzz, which has a strong track record of bug finding thanks to an investment of significant computational resources. The downside is that fuzz testing is a game of diminishing returns, where even more computation resources will be required for ever smaller gains.

That said, the practical benefits of fuzzing clearly outweigh the risks of no fuzz testing at all. The track record in terms of bugs and vulnerabilities discovered means fuzz testing ought to remain the cornerstone of modern software testing best practice.

SWOT Analysis for Fuzzing

Strengths	**Weaknesses**
● Fuzzing is a practical method for real world bug detection ● Proven to be effective, with a strong track record ● 1000s of discovered CVEs attributed to fuzzing ● Superior coverage than traditional software testing methodologies ● Fuzzing is scalable, and possible with limited resources	● Fuzzing may not detect bugs due to incomplete coverage, which is a fundamental problem with software testing. ● Fuzzing is resource bound, with diminishing returns ● Some fuzzing tools have poor usability
Opportunities	**Threats**
● Fuzzing can be extended to other domains, e.g., IoT, DNNs, search-based algorithms ● Algorithmic improvements to fuzzing can enhance both speed and coverage ● Many ongoing research opportunities related to fuzz testing	● Poor adoption of fuzzing techniques may introduce cybersecurity risk

Program Repair and Binary Hardening

Software systems are prone to vulnerabilities that are open to exploitation by malicious third parties. One of the key difficulties in building trustworthy software systems is the lack of specifications (i.e., a description of the intended behavior), meaning that it is difficult to determine whether a specific behaviour is benign or malicious. A specification discovery process can help fill the void, allowing for automatically generated repairs or hardening of program behaviour—and moving closer to the goal of self-healing software systems. As more and more of our daily functionalities become software controlled, and with the impending arrival of new technologies such as personalized drones, the need for self-healing software has never been greater.

Binary software presents some additional complications. Software components are often distributed without source code in order to protect intellectual property rights and to discourage reverse engineering. Binary-only components, including *Commercial Off-the- Shelf* (COTS) software, are ubiquitous. These can range from everything including closed-source programs, libraries, as well as embedded firmware in IoT and networking hardware. COTS software is part of most modern software stacks, including those used by critical infrastructure, such as 5G mobile networks. COTS software may be untrusted, contain security vulnerabilities, or may be the victim of malicious interference such as *supply chain attacks*. The lack of source code makes binary rewriting/repair very challenging, since the information necessary for specification recovery has often been stripped from the binary code. Furthermore, even if the source code is available, typically a binary-only component will be used, with the implied trust that the binary is derived from the source. Finally, the directl patching or repairing binary-only code is well known to be a very difficult problem, and may introduce bugs or other compatibility issues on its own.

The core NSoE-TSS team and grant recipients have been developing technologies to help secure untrusted source and binary components through a combination of analysis, hardening and repair.

How does binary hardening and repair help?

Software hardening and repair are established methods for mitigating the potential impacts of untrustworthy software components used by modern software systems. Such software systems can be impacted by both undiscovered bugs/vulnerabilities, or existing (known) bugs or vulnerabilities for which a patch does not exist or cannot be applied. For many software components, the source code is unavailable, e.g., with closed source or *commercial-off-the-shelf* (COTS) software. This means that traditional mitigations, such as source-level static analysis techniques, are not possible.

Binary hardening and repair helps developers and users mitigate the risks of software bugs and vulnerabilities in modern software systems that use closed-source or COTS components.

Binary hardening can be applied so that potential (unknown or undiscovered) vulnerabilities can be detected before harm is caused. For example, a binary can be hardened against memory errors (e.g., buffer overflows or use-after-free errors), meaning that such errors can be detected before they can be exploited. For example, the error may be logged and the program immediately terminated.

Similarly, binary repair can be used to fix known discovered bugs at the binary-level. Unlike traditional *Automated Program Repair* (APR) techniques, binary repair can work without knowledge of the source code. This can be useful if a binary-only software component has a recently discovered vulnerability (e.g., a zero-day), but the developer patch is not yet available. It can also be useful for legacy and unmaintained software, where no further development on the software is possible, but the component is nevertheless relied upon.

What are the challenges in binary hardening and repair?

There are several challenges in the translation of binary hardening and repair technologies.

How can closed-source (binary-only) software be trusted?

Software is commonly distributed in binary form, meaning that the source code is not open to inspection. analysis or scrutiny. The users of the software must essentially trust that it is non-malicious, and/or that it does not contain some vulnerability that can be exploited by some third-party. Experience has shown that these assumptions do not hold in practice, and it is common for security holes to be found in binary-only software, either days, months or even years after release. The problem can be compounded by the lag between when vulnerabilities are first reported versus when the bug is fixed. During this delay, any software system relying on vulnerable components can be open to exploitation.

For example, unsafe programming languages, such as C/C++ remain very popular. For efficiency, these languages use *manual memory management* by default, meaning that it is up to the developer to correctly manage the allocation and access to objects. If the developer makes a mistake, then a *memory error* will occur. Memory errors are useful to attackers since they can be used to modify the contents of memory, and be used as the basis for various attacks, including denial-of-service, control-flow-hacking, and information disclosure. Detecting memory errors at the source-code level (either through static or dynamic analysis) is difficult. Memory error detection at the binary-level is even more difficult.

Binary code may also be vulnerable to intentionally malicious behaviour. Recent examples of this problem have manifested in the form of *supply chain attacks.* Here, a software system is composed of several components, where some of these components are supplied by presumably trusted third parties. Here, an attacker attempts to inject an exploit into the target software system via one of these components. In a typical example, the attacker makes a malicious version of a binary available in popular code repositories, with the hope that software

developers will use and integrate it into their systems. The problem is amplified when the components are binary-only, and may be blindly trusted without any additional scrutiny.

Possible Solutions: One possible solution is to foster a culture of not blindly trusting binary components, and taking mitigations when necessary. Mitigations can include binary testing, fuzzing, analysis, hardening and repair. Specifically, fuzzing/analysis may help detect vulnerable or malicious behaviours, hardening may protect against undiscovered malicious behaviors, and repair may be used to fix known/discovered problems. The adoption of one or more of these methods may help mitigate problems caused by the blind trust of binary-only software components.

For this to be feasible, suitable binary-only tools must be made readily available to software developers. However, there are several well-known impediments to wider adoption of binary analysis techniques, mostly related to the difficulty with the analysis and rewriting on binary code. Any tools developed along these lines must have a high degree of reliability before they are likely to enjoy wider adoption. This has been the main achilles heel of wider adoption of binary-only methods, and leads directly into our second challenge question.

How can we make binary analysis, rewriting and repair more reliable/scalable?

One of the main technical challenges in dealing with binary code is the complexity. Unlike source-level software, binary code is typically missing critical information, such as symbols or types, that are usually necessary for typical rewriting/repair applications . The traditional solution is to use binary analysis in order to attempt to recover this information directly from binary code. However, most forms of binary analysis necessarily rely on heuristics or assumptions, since the problem of static binary analysis is well known to be theoretically unsolvable in the general case. The reliance on assumptions or heuristics will often lead to inaccuracies, which may propagate up the tool chain (e.g., if an inaccurate binary analysis result is relied on by another tool), leading to incorrect or incomplete results.

For example, the traditional methods for binary rewriting rely on the recovery of the program's *control flow*, such as jumps, function calls and type information, so that the program's structure can be reconstructed in the rewritten binary. However, the accurate recovery of control-flow information is not possible in the general case, meaning that the resulting rewritten binary may not be correct. If the binary rewriting tool is used as the foundation for some other application, such as binary hardening or repair, then such errors will manifest. While the accuracy of binary function signature recovery has seen some improvement [FSROPT], the problem still remains challenging in the general case.

These problems can greatly impact the utility of the intended application. For example, the aim or binary hardening or repair is to mitigate or eliminate bugs in existing binary code. However, if the binary rewriting is inaccurate, this process itself may introduce new bugs, which defeats the original purpose. These problems mean that binary-only tools and methodologies have not enjoyed more widespread adoption.

Possible Solutions: The NSoE-TSS has been pushing the development of binary analysis and rewriting methods that do not use analysis, thereby eliminating a possible source of errors. A key example of this is the E9Patch binary rewriting system [E9Patch], which develops a binary rewriting methodology for the x86_64 that does not rely on control-flow information. Other examples include dynamic binary rewriting and translation technologies, such as DynamoRIO and QEMU, that do not need to rely on static analysis.

Some applications can tolerate approximate binary analysis. For example, binary repair applications require analysis to determine a suitable “fix location” which indicates where the patch should be applied. Accurately determining the fix location for a given bug cannot be solved in the general case. However, since the patched binary will be tested and evaluated by a developer/user anyway, perfect accuracy can be substituted for a practical analysis that can quickly eliminate large numbers of irrelevant locations. This has led to the development of tools such as VulnLoc under the NSoE-TSS. Here, VulnLoc uses concentrated fuzzing to quickly narrow the set of possible patch locations.

We believe that analysis-free (or at least analysis-tolerant) tools that work on binaries are the most promising moving forward. However, this is limited to what is technically feasible.

How can binary-only tools become more usable?

Tools for working on binary-only code are typically designed for experts in the field. This is especially true since the average software developer only has limited exposure to low-level programming, especially in assembly. Binary software is not designed to be human readable, and often these complexities are reflected in tools designed to work with binary code.

Possible Solutions: One approach is to design tools that are fully automatic, meaning that no experience with binary code is necessary in order to use the tool. This is feasible for some applications. For example, the NSoE-TSS has developed the E9AFL tool for automatically instrumenting binary code with AFL (fuzzer) instrumentation. The tool is ”push-button”, meaning that the user simply inputs a binary, and the tool outputs a new binary with AFL instrumentation applied. The tool is even more convenient to use than traditional (source-level) instrumentation methods, since there is no need to recompile the program with a special compiler toolchain (afl-gcc).

We believe this approach can be extended to other applications. For example, binary hardening tools can also be push-button in principle. Another example is *Automated Program Repair* (APR), where tools generate path candidates that attempt to resolve a known bug discovered in source code. However, applying a patch requires recompilation of the program, which can be a slow process. With binary rewriting, it may be possible to directly patch the program at the binary level, meaning that recompilation is not required. This is more convenient to the end user, for similar reasons as to why E9AFL can be more convenient than traditional tools that rely on recompilation.

How can we bring binary-only technologies into other domains?

The most mature binary rewriting, repair and hardening technologies have been developed for the x86_64, and most notably for Linux. However, there are other sizable domains that ought to be considered, such as other *Instruction Set Architecture* (ISA) including ARM, as well as other operating system environments such as Windows. *Internet of Things* (IoT) and embedded software is another domain that has received significant attention in recent years. Currently, there is a gap between the tools/technologies developed for one domain versus others, and this leads to a lack of capability within these domains.

Unfortunately, the problem is not as simple as a lack of manpower to develop tools. Many binary-only technologies are highly platform-specific, meaning that it is not as simple as porting an existing tool to another system. For example, it is often the case that an underlying binary rewriting technology is ISA-specific, meaning that a technology developed for x86_64 cannot simply be ported to ARM64. For example, the E9Patch tool developed by the NSoE-TSS is x86_64 specific.

Another challenge is the diversity of environments, especially in the IoT domain. Here, it is common for IoT software to be very low-level, and be intimately tied to specific hardware (e.g., sensors or controllers). The software is often closed-sourced and embedded in ROM, meaning that it is difficult to access and modify. Even if accessed, perfectly emulating the hardware environment can be difficult.

Possible Solutions: For some domains, emulation such as that using systems such as QEMU may be the only viable solution. However, this may be unsatisfactory for some applications.

Another solution is to diversify the base of existing tools, even if each individual tool is highly specialized. For example, although tools such as E9Patch cannot be ported directly to ARM, it may be possible to create an ARM-specific tool that provides a similar interface and/or capability under a different platform. This however requires a manpower commitment, and may be limited to what is technical.

What are the risks with binary hardening and repair?

The main risk with anything related to binary rewriting is how reliable the user-level tools really are. As noted for the application on binary hardening, the aim is to mitigate existing bugs rather than introduce new bugs. Tools that only partially work or work on limited applications are unlikely to enjoy widespread success, and will result in poor adoption. The problem is also especially important for safety critical applications, where any binary analysis or rewriting error may have serious implications.

Possible Solutions: The NSoE-TSS recognises these risks and has specifically developed technologies designed for scalability and soundness over other factors, such as raw performance. This is evidenced by the E9Patch binary rewriting system, which uses binary rewriting methods that are optimized for soundness over speed. This is demonstrated by the fact that E9Patch is the only binary rewriting system that can successfully rewrite very large programs such as Google Chrome.

The risk can be further mitigated by augmenting the process with traditional software quality practices, such as fuzzing or formal verification. Both of these are also core NSoE-TSS activities. Fuzzing with E9AFL also indirectly tests the correctness of the underlying binary rewriting process.

SWOT Analysis for Binary Analysis

Strengths	**Weaknesses**
● Binary methods are permissionless, and be applied to untrusted closed source or COTS binaries ● Binary methods can be applied to legacy systems without source code ● Binary methods do not require recompilation which can streamline workflows	● Binary analysis without source code is difficult due to missing information ● Binary rewriting can introduce errors which affects the utility of the solution ● Binary analysis/patching can be a “blackbox”, and difficult for developers to understand or verify ● Binary methods typically have higher overheads than equivalent source-level solutions
Opportunities	**Threats**
● Recent advances in binary rewriting make it applicable to larger binaries ● Many source-level applications can be ported to equivalent applications at the binary-level	● Inaccurate binary analysis can introduce more bugs than are solved ● The additional overheads of binary methods may be unattractive.

Artificial Intelligence Testing

Artificial Intelligence (AI) has achieved tremendous success in many applications. However, similar to traditional software, AI models can also contain defects, suAI models can also contain defects, such as adversarial attacks, backdoor and noisy labels.ch as adversarial attacks, backdoor and noisy labels. Quality assurance of AI systems is needed. Some mature quality assurance techniques have been developed for traditional software (e.g., fuzzing, hardening, repair). However, they could not be used in the quality assurance of AI systems due to the fundamental differences between AI systems and programs.

Quality assurance of AI systems has been a hot topic in AI, security and software engineering research communities. The common way of evaluating the quality of the AI models is based on the collected testing dataset. However, the dataset may be biased, which could overestimate the quality of the model. Thus, the systematic model evaluation and enhancement are required for building trustworthy AI systems. Different techniques have been proposed such as adversarial attacks, testing and robustness enhancement, which is still at an early stage.

Artificial Intelligence testing is one of the core competencies of the NSoE-TSS project towards building trustworthy AI systems. The core NSoE-TSS team and grant recipients have been developing technologies to help evaluate and enhance the trustworthiness of AI systems.

How does AI quality assurance techniques help?

AI quality assurance techniques are proposed for improving the quality of the AI systems. Different from the techniques for traditional software, AI quality techniques consider the unique characteristics of the AI systems, i.e., they are mainly data-driven. The AI systems usually contain the data part and the model part. The proposed quality assurance techniques can help improve the quality of the training data and the trained models.

Specifically, NSoE-TSS project will develop the techniques including runtime monitor, repair, noisy label handling and adversarial example detection. These techniques cover the main lifecycle of AI systems including the development and deployment phases. The noisy label handling is developed to detect the noises in the training data. The high-quality data is quite important for developing a high-quality model. The repair, runtime monitor and adversarial example detection techniques are developed to guarantee the robustness of the system when it is deployed in the specific environment. The developed techniques in this project can be applied into different applications in smart nation infrastructures, e.g., image classification, audio recognition, autonomous driving, healthcare, finance, etc.

What are the challenges in building trustworthy AI systems?

There are several challenges in the AI quality assurance techniques:

How can we know the model makes the trustworthy decision on runtime?

The widespread adoption of Deep Neural Networks (DNNs) in important domains raises questions about the trustworthiness of DNN outputs. Even a highly accurate DNN will make mistakes some of the time, and in settings like self-driving vehicles these mistakes must be quickly detected and properly dealt with in deployment. Just as our community has developed effective techniques and mechanisms to monitor and check programmed components, we believe it is now necessary to do the same for DNNs.

Possible Solutions: One possible way is to check the DNN as a process by which internal DNN layer features are used to check DNN predictions. We could build a self-checking system that monitors DNN outputs and triggers an alarm if the internal layer features of the model are inconsistent with the final prediction. Moreover, we could provide the advice in the form of an alternative prediction. By this way, when the system is deployed, we can perform the runtime checking such that the decision is more trustworthy.

How can we know the model can work well on the unexpected runtime input/sample?

Trained with a sufficiently large training and testing dataset, Deep Neural Networks (DNNs) are expected to generalize. However, inputs may deviate from the training dataset distribution in real deployments. This is a fundamental issue with using a finite dataset. Even worse, real inputs may change over time from the expected distribution. Taken together, these issues may lead deployed DNNs to mis-predict in production.

Possible Solutions: One possible way is to mitigate DNN mis-predictions caused by the unexpected runtime inputs to the DNN. We can treat the DNN as a blackbox and focus on the inputs to the DNN: 1) we first recognize and distinguish “unseen” semantically-preserving inputs. A distribution analyzer (e.g., based on the distance metric) can be. 2) Then we could transform those unexpected inputs into inputs from the training set that are identified as having similar semantics. This process can be formulated as a search problem over the embedding space on the training set. This embedding space is learned by a network as an auxiliary model for the subject model to improve the generalization.

Another solution is to monitor the outputs of DNNs to detect irregularities. To this end, the NSoE-TSS has developed *SelfChecker* [SelfChecker], a self-checking system that will trigger an alarm if the internal layer features of the model are inconsistent with the final prediction.

How can we improve the adversarial robustness of pre-trained classifiers?

A significant challenge is how to defend adversarial attacks on pre-trained classifiers. Deep neural networks are known to be vulnerable to adversarial attacks, where a small perturbation leads to the misclassification of a given input [Pure]. In real applications, many existing neural networks used in industries and website services are trained without considering the adversarial robustness, thereby being vulnerable to adversarial attacks.

There are several challenges in improving the adversarial robustness of pre-trained classifiers. On one hand, it is too expensive and impractical to re-train all the existing neural networks used in real applications. Therefore, it is important to provide adversarial defense for pre-trained models without any additional training or fine-tuning. On the other hand, we may not have access to the original model. For example, a user of a public version API might want to use the API to create robust predictions, but may not have access to the underlying non-robust model. Existing adversaries on provable robustness mainly focus on training classifiers from scratch, we consider the problem of generating a robust classifier without retraining the underlying model at all.

Possible Solutions: To improve the adversarial robustness of pre-trained classifiers, a possible solution is to simply apply an off-the-shelf “filter” that would allow practitioners to automatically generate a robust model from a standard model.

For example, a custom-trained denoiser can be prepended before the pre-trained classifier. When queried at a point x, the smoothed classifier outputs the class that is most likely to be returned by the original model under isotropic Gaussian perturbations of its input. However, randomized smoothing requires that the underlying classifier is robust to relatively large random Gaussian perturbations of the input, which is not the case for off-the-shelf pre-trained models. Through the custom-trained denoiser, we can then effectively make the model robust to such Gaussian perturbations, thereby making it “suitable” for randomized smoothing.

The robustness of the image classifier is just an example. The NSoE-TSS also considers the case in real-world physical adversarial perturbations, where it would be much more challenging to improve the adversarial robustness of pre-trained classifiers.

How to handle unrestricted physical adversarial attacks?

Another challenge is to defend physical adversarial attacks that are not restricted in the visibility of the perturbations. Existing adversarial defense algorithms are mostly focused on applying the attacks in virtual applications and controlled datasets, where the attacks are bounded with l2 norm or lp norm to make them invisible. However, in the physical adversarial attacks, many distortions that are normally observed in a common scene are not restricted in the visibility of the perturbations.

There are several main challenges in considering unrestricted physical adversarial attacks. On one hand, it would be more difficult to simulate unrestricted adversarial attacks than bounded adversarial attacks due to the increase of the search space. Without constraints in the attack strength, it is also hard to define the adversarial attacks which would not be recognized by humans.

On the other hand, large adversarial attacks may make it infeasible to be defended by existing adversarial training methods. Simply combining perturbations with different lp norms in adversarial training has been proven to be useless in the generalization of unseen attacks. Though significant, the generalization problem of adversarial training on unrestricted physical adversarial attacks is only occasionally studied at this time. One possible reason is that our understanding of adversarial examples is limited and incomplete.

Possible Solutions: to handle unrestricted physical adversarial attacks, a possible solution is augmenting the simulated adversarial attacks during training, such as increasing the number and diversity of targeted models and adversarial attacks.

For example, the training data can be augmented with perturbations transferred from various models and produced by different attack algorithms. In such an ensemble style, the simulated adversarial attacks would be more diversified, extending the generalization of neural networks to more unseen attacks. Besides, different lp norms can also be applied to increase the diversity of the adversaries. Using GAN to explicitly model the distribution of adversarial examples around each data point is also a potential method to define the physical adversarial attacks in real-world applications.

How to validate the performance of dynamically generated neural networks for counteracting adversarial examples?

In the scheme of moving target defense (MTD), a dynamic ensemble consisting of multiple neural networks can be generated to counteract the adversarial-example attacks. The rationale of this defense is that, since the construction of effective adversarial examples requires the task neural networks in either whitebox or blackbox setting, the adversary's construction process can be largely impeded if we prevent the adversary from obtaining the task neural networks being used. The existing research has shown that the HyperNet is a useful technique to speed up the generation of the dynamic ensemble, which strengthens the MTD security. However, the dynamics of the task ensemble creates two concerns in the real-world deployments:

- How do we validate or predict the performance of a newly generated ensemble in real time?
- HyperNet-generated neural networks in general have lower performance than those directly trained from the training data. Are there technical approaches to further improve the performance of HyperNet-generated neural networks?

Possible solutions: A straightforward way to validate or predict the performance of a newly generated ensemble is to use a validation dataset that is carried by the autonomous system.

However, the data-driven validation process may introduce long time delays and negate the benefit of using HyperNet for fast ensemble generation. A pragmatic approach to shorten the validation delay is to leverage the parallel processing capability of the autonomous hardware accelerator. In addition, another possible approach is to design a deep neural network that takes the ensemble's neural network weights as input and outputs the predicted performance of the ensemble. The intuition behind is that since the performance of a neural network depends on its weights, it is possible to train such a performance predictor specific to a HyperNet.

Once the performance predictor mentioned above can be trained, we can use it to screen the HyperNet-generated neural networks and select those that are predicted to give the best accuracy. Since the generation and selection of the neural network is subject to a soft deadline, some scheduling algorithm can be developed to control the selection process such that the average image processing throughput over a time period meets the requirement.

Challenge 2: As HyperNet was not originally designed under an adversarial setting, it is challenging to improve HyperNet capability to counteract highly adversarial attackers.

HyperNet was originally proposed to improve the inference accuracy since it can generate multiple neural networks to form an ensemble. However, under the adversarial setting, if the attacker has obtained the deployed HyperNet, the attacker can also improve its adversarial examples to increase attack success rate.

Possible solutions: Adversarial learning and federated learning may be adopted to address the challenge. First, in the adversarial learning scheme, several adversarial example construction algorithms can be employed as the opponent side to improve the training of the HyperNet. As a result, it imposes more challenges for the attacker to generate effective adversarial examples using existing construction algorithms. So far, there is no integration of HyperNet and adversarial learning. Challenges in adversarially training the HyperNet can be expected. More research is needed to implement this.

Second, if we can diversify the HyperNets at the autonomous systems, the attacker who has compromised one autonomous system may not easily transfer to another autonomous system. Federated learning may be applied. During the federated learning process, a coordinator orchestrates the weight updates at distributed autonomous systems. Differences among the autonomous systems' HyperNets can remain.

SWOT Analysis for AI quality assurance

Strengths	**Weaknesses**
● Provide the analysis on the physical attacks ● The methods can be used in the deployment phase that does not require the modification of the original model. ● The methods are general and can be easily extended in different applications.	● It is difficult to enumerate all kinds of attacks. ● The runtime monitor may provide incorrect advice for some inputs ● It can be difficult to apply in black-box level DNNs without knowing the architecture of the models
Opportunities	**Threats**
● More AI applications will be deployed in the future. ● We can collect diverse attacks and erroneous inputs when the system is deployed, which can further improve our techniques.	● Improper use of repair and runtime recommendations may provide incorrect decisions or may be exploited by the attackers.

Formal Verification

Formal Verification is an established technique for formally proving the correctness of a system. While other techniques only explore part of the domain under analysis, making it inefficient especially for concurrent systems, formal verification usually aims to cover the entire state space. One well-known verification technique is model checking, which automatically explores the state space of a model representing the system to check if it is correct. However model checking often does not scale well for large systems. Instead, it is possible to use deductive verification, which uses theorem provers to generate proof obligations that can be proven automatically often with the help of SMT solvers, or in complex cases, guided by experts. While deductive verification is more expensive, it is especially suited for the verification of high assurance systems, where their correctness is critical.

Formal Verification is a core competency of the NSoE-TSS project. In particular, formal verification in the NSoE-TSS focuses on three different aspects: first, the automation of deductive verification of concurrent systems, especially suited for micro-kernel verification; second, procedures to guarantee the memory safety of programs; and third verification of distributed systems.

How can formal verification help?

Formal methods provide foundations for the precise and unambiguous description of complex systems, and to reason on properties that they must satisfy. Other techniques like testing or fuzzing, can only provide the cases in which a system is not failing, but they cannot provide any evidence out of the cases they were designed to test. Formal methods, by providing proof of correctness, guarantee that the expected behaviour is fulfilled by the system.

Formal methods aim to fully explore the state space of a system to ensure that its security and functional properties are preserved for every possible behaviour of the system. Guaranteeing that the security and correctness of a system are preserved in every execution trace is essential on critical systems.

The importance of applying formal methods to the development of a system does not only lie down to formal verification, but also to formally specifying the system under analysis. A formal specification uses a formal description language to provide a rigorous and non-ambiguous representation of the system, which helps to identify inconsistencies and problems in the early stages of the development.

When focussing on formal verification, automatic verification uses fully automated tools (also known as push-button verification) in which users only have to specify the system behaviour and the desired properties in the description language the tool uses. Theorem proving verification uses theorem proving to show the correctness of the theorems describing the

properties that the system must preserve. Using theorem proving requires a deeper knowledge of the system under analysis, and the verification process requires a larger amount of manpower since often the theorem proving process is not fully automated but guided by users. Because of its higher cost, this technique is used for the verification of critical systems for which total assurance is required by certification standards like Common Criteria and DO-178B.

Although automatic verification has been the preferred approach for the verification of systems, the theorem proving approach is becoming more popular thanks to the development of more powerful theorem proving tools and logics, which make the verification of real-world systems.

In particular, formal verification of concurrent systems becomes the only feasible option for ensuring the correctness of a system. The possible execution traces on concurrent systems grows exponentially to the number of concurrent processes, making other techniques ineffective.

What are the challenges in formal verification?

There are some challenges to solve when applying formal methods to the verification of systems.

How can we make formal verification more scalable?

The main problem we face when applying formal verification for the correctness of systems is to make it scalable to large and complex systems. Push-button automatic techniques exploring the whole system state space suffers from the unfamous *state-space explosion* problem. The size of the state space of a system grows exponentially to the number of state variables and hence the required processing resources to carry out the verification. This problem greatly limits the application domains for automatic verification techniques, not being able to cope with the number of states that large systems generate.

State-space explosion is not a problem when following the guided theorem proving verification approach. In this case, the verification is not carried out by explicitly exploring the system state space, but rather inductively on the data and language structures. While not suffering from state space explosion, scalability of the verification by using theorem proving can be increased by decreasing the necessary effort to carry out the verification.

Possible Solutions: Some techniques are used to mitigate the state space explosion, like symmetry reduction, partial order reduction, state abstraction, and symbolic execution. By using these techniques the verification of larger systems is possible, but verification of very large complex systems is still not possible and it is necessary to use the more expensive and powerful theorem proving approach.

When following the theorem proving approach, it is possible to provide specific logics that can be applied to the problem under verification. It is also possible to increase the automation of the theorem proving by implementing decision procedures that automatically applies the reasoning framework that the logic provides To that aim NSoE-TSS is focussing on the development of logics for the verification of concurrent systems that can be applied to the verification of concurrent specifications. In NSoE-TSS we are also developing automatic methods to apply the concurrent logic to reduce the effort required to the verification of concurrent specifications.

The NSoE-TSS has also developed scalable technologies towards practical applications, such as the *CoSplit* [CoSplit] analysis tool that soundly infers ownership and commutativity summaries for arbitrary smart contracts

How can we make formal verification more usable?

Automatic verification only requires providing the properties that the system must preserve, since it is possible to automatically translate programs source code to the formal description technique the verification tool uses. The specification of properties is done by using temporal logics that have a learning curve not much different to that of programming languages.

Possible Solutions: Theorem proving verification requires a high expertise level on the proving tool and the logics used for the specification of the properties and the system itself. NSoE-TSS has been focussing on the development of CSimpl, a specification language similar to system languages that allow embedding into the theorem prover implementation-like specifications and hence simplifying the modelling of specifications. Also, in NSoE-TSS we have developed an automatic procedure for the translation of concurrent C source code into *CSimpl* [CSimpl] specifications. The decision procedures created in NSoE-TSS eases the reasoning process, simplifying the guided verification in the theorem prover.

SWOT Analysis for Formal Verification

Strengths	**Weaknesses**
● Formal verification guarantees the correctness of a system with regards a set of desired properties ● Guided theorem proving can be used for the verification of any potential system. Including infinite systems. ● Formal Methods can be used in the early stages of the design to detect inconsistencies.	● Formal verification requires the source code for the verification of software. ● Automatic verification may not scale for large systems ● Theorem proving verification requires a deep knowledge on the application and the theorem prover.
Opportunities	**Threats**
● Recent advances on guided theorem proving add more automation simplifying the proving procedure ● Many research opportunities on formal verification to increase the automation of theorem proving verification	● The cost of guided theorem prover verification may be unattractive, especially for non-critical systems

Bibliography

[E9Patch] Gregory J. Duck, Xiang Gao, Abhik Roychoudhury, *Binary Rewriting without Control Flow Recovery*, Programming Language Design and Implementation, 2020
[E9AFL] Xiang Gao, Gregory J. Duck, Abhik Roychoudhury, *Scalable Fuzzing of Program Binaries with E9AFL*, Automated Software Engineering, 2021
[VulLoc] Shiqi Sheng, Aashish Kolluri and Zheng Dong and Prateek Saxena, Abhik Roychoudhury, *Localizing Vulnerabilities Statistically From One Exploit*, Asia Conference on Computer and Communications Security, 2021
[OASIS] Jiaqi Hong, Xuhua Ding, *A Novel Dynamic Analysis Infrastructure to Instrument Untrusted Execution Flow Across User-Kernel Spaces*, Security and Privacy, 2021
[Sensei] Xiang Gao, Ripon K. Saha, Mukul R. Prasad, Abhik Roychoudhury, *Data Augmentation to Improve Robustness of Deep Neural Networks*, International Conference on Software Engineering, 2020
[FSROPT] Yan Lin, Debin Gao, *When Function Signature Recovery Meets Compiler Optimization*, Security and Privacy, 2021.
[Fix2Fit] Xiang Gao, Sergey Mechtaev, Abhik Roychoudhury, *Fix2Fit: Crash-avoiding Program Repair*, International Symposium on Software Testing and Analysis, 2019
[CPR] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, Abhik Roychoudhury, *Concolic Program Repair*, Programming Language Design and Implementation, 2021
[TimeMachine] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury, *Time-travel Testing of Android Apps*, International Conference on Software Engineering, 2020
[DeepHunder] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, Simon See, *DeepHunter: a Coverage-guided Fuzz Testing Framework for Deep Neural Networks*, International Symposium on Software Testing and Analysis, 2019
[Pure] Lei Feng, Jiaqi Lv, Bo Han, Miao Xu, Gang Niu, Xin Geng, Bo An, *Provably Consistent Partial-label Learning*, Masashi Sugiyama. Neural Information Processing Systems
[SelfCheck] Yan Xiao, Ivan Beschastnikh, David S. Rosenblum, Changsheng Sun, Sebastian Elbaum, Yun Lin, Jin Song Dong, *Self-Checking Deep Neural Networks in Deployment*, International Conference on Software Engineering, 2021.
[CSimpl] David Sanán, Yongwang Zhao, Zhe Hou, Fuyuan Zhang, Alwen Tiu, Yang Liu, *CSimpl: A Rely-guarantee-based Framework for Verification of Concurrent Systems*, Tools and Algorithms for the Construction and Analysis of Systems, 2017
[CoSplit] George Pîrlea, Amrit Kumar, and Ilya Sergey, *Practical Smart Contract Sharding with Ownership and Commutativity Analysis*, Programming Language Design and Implementation 2021
[FixMorph] Ridwan Shariffdeen, Xiang Gao, Gregory J Duck, Shin Hwei Tan, Julia Lawall, Abhik Roychoudhury, *Automated Patch Backporting in Linux (Experience Paper)*, International Symposium on Software Testing and Analysis, 2021